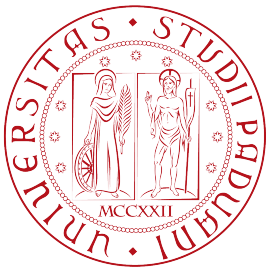

UNIVERSITÀ DEGLI STUDI DI PADOVA

DEPARTMENT OF INFORMATION ENGINEERING

Master Course in COMPUTER ENGINEERING

Master Course Thesis

Keyword Search in Relational Databases: Architecture, Approaches and Considerations

Supervisor:

Prof. Maristella AGOSTI

Graduate:

Andrea CARRARO

Co-supervisor:

Dr. Gianmaria SILVELLO

February 21, 2017

Academic Year 2016-2017

Abstract

Through the years, keyword search established as the most common and easy way to retrieve information from document collections and on the web. However, a large amount of data stored in structured databases cannot be accessed due to the difficulty of posing natural language queries to structured sources such as relational databases. Many systems have been proposed to apply keyword search to relational databases, in order to allow them to be accessed and consulted by general users. This thesis investigates the solutions presented in the literature and proposes a general pipeline to design and develop keyword search systems. We survey the different approaches proposed by the community focusing on the different elements that compose the system pipeline: we analyze the data representation and query processing components, the mechanisms to efficiently locate relevant database elements and the algorithms to build the answers fulfilling the user information need. Finally, we make the point on the evaluation of these systems with reference to the well-established and widely used Cranfield framework adopted in the Information Retrieval field.

Sommario

Negli ultimi anni il paradigma *keyword search* si è imposto in modo importante nell'ambito del reperimento di informazioni in collezioni di documenti e nel web, merito soprattutto della sua facilità d'uso. Nonostante ciò, le informazioni contenute nei database strutturati, come ad esempio i relazionali, ne rimangono escluse a causa dell'impossibilità di interrogare queste sorgenti usando il linguaggio naturale. Questo lavoro di tesi presenta le diverse soluzioni proposte in letteratura per applicare il paradigma keyword search alle basi di dati relazionali, e vuole delineare una architettura generale per definire e sviluppare questi sistemi. A tal proposito, le soluzioni presentate dalla comunità scientifica sono state analizzate focalizzandosi sui singoli componenti della pipeline di ricerca, quali la rappresentazione dei dati, l'elaborazione della query, il processo di abbinamento delle keyword con gli elementi del database, e la costruzione delle risposte da fornire all'utente. Infine, si sono analizzati i processi di valutazione sperimentale di questi sistemi, prendendo come riferimento l'ormai consolidato paradigma di Cranfield, universalmente adottato nel campo del Reperimento dell'Informazione.

Contents

1 Introduction

In our modern and connected word, the keyword search paradigm has become the preferential way to search and retrieve information on documents collections or on the world wide web. The use of web search engines like Google or Bing educate the users to pose natural language queries defining their information need in a simple and straightforward way, and to critically inspect the results page in order to search for an answer fulfilling the need. Keyword search has been extensively studied in the *Information Retrieval* (IR) field since the 1960s. The main focus of IR is the study of the techniques and processes to implement and enhance automated systems that, taking as input a keyword query, return a list of documents, ranked by a measure of their relevance to the query according to a certain ranking function. The documents are essentially "bags of words", unstructured document containing plain text to be indexed, and generally do not contain links and connections among them, so that a document could be seen as a logical information unit.

Typically, traditional keyword search paradigm has been applied only to document collections, whereas a vast quantity of data are stored in

structured sources such as relational databases. These data are accessed through *Structured Query Language* -SQL- queries, that present two critical aspects:

- The user must pose queries using a specific syntax accordingly to the query language adopted by the database management system.

- The user must know the *schema* of the underlying database in order to issue a query .

The research on applying keyword search to the databases field has the aim to design and develop systems providing the possibility to freely and naturally access relational database contents by the means of *natural language*. To apply a natural language query to a structured database means to handle the query ambiguity, by providing all the answers for all the possible query interpretations. For example, we assume the input of a keyword search systems to be the query {Search, Hristidis}, posed to a scientific literature database. The system does not have any clues about the user information need, but it finds that the term “Search” appears in the *articles* relation and the term “Hristidis” in the *authors* relation. The system does not know if “Hristidis” is meant to be the name of the author of an “article” containing the word “Search” or if in an article “Hristidis” is cited, so it must return all the relative answers. It is then to the user to critically chose the most pertinent results or to rephrase the query to obtain better results. This approach is the contrary of the usual database consultation where the user issues structured and not ambiguous queries which always return the "right" and complete answers.

In the keyword search for structured data case, differently from what happens in the IR field and for web searches, the answers returned are not plain documents or webpages, but data structures built from the data contained in the database, starting from the elements (relations, column or tuples) that contains the query keywords, i.e. that *match* a keyword. These answers present different structures, depending on the approach used by the system.

Building these structures is a hard task in terms of algorithmic complexity, so that most of the research in the keyword search area is focused on enhancing the performance and efficiency. For this reason, as described in [Yu et al., 2010], keyword search in databases is completely different from traditional IR, because the first focuses on the interconnected object structures, whereas the second focuses on the objects content.

In the last years, a lot of effort has been put to define new systems and algorithms to efficiently implement keyword search in databases. Two main approaches have been proposed in the literature, namely the schema- and the graph-based approach. In the schema-based approach the database is represented as a graph of connected database relations, while the graph-based solutions model the nodes as tuples. In these approaches, the answers are built on-the-fly after the query has been issued, connecting the nodes containing the keywords by means of the graph edges, which represent the primary/foreign key constraints of the database.

The poor time performances of the schema- and graph- based systems encouraged the community to propose systems that precompute virtual documents from the information stored in the database, in order to efficiently retrieve these documents with traditional IR techniques. Even

if the *virtual document approach* did not received the same attention of the other approaches, the solution to precompute part of the workload could be a key-element for the future of this research area.

Apart form the specific approach, each keyword search system realizes a general pipeline to compute and return to the user the appropriate results

- The system prepares the environment to allow the search, i.e. it builds the indexes and the auxiliary structures needed for the algorithms.
- The system processes the keyword query and match each keyword (or a subset of the keywords) to the database elements, such as relations, columns or tuples.
- The matched elements must be connected in order to build meaningful answers according to the query and the chosen approach, relying on the appropriate graph representation.

This work aims to present to the reader a general and unified architecture to design a keyword search in databases system. Following the pipeline outlined above, we survey the different solutions proposed to describe each architectural component, according to the peculiarities of the different approaches. We analyze the advantages and disadvantages of each solution, the time and memory consumption and the effectiveness of the search, in order to provide a complete and objective overview on the state-of-the-art of this field. The considerations that we make in this work are meant to help a further research on this area, so that new systems and approaches could be designed by taking into account the work already done by the community and the issue arisen, overcoming the limits of the current generation of keyword search systems.

In Chapter 2 we provide the reader with the necessary background to understand the contents of the survey, unequivocally defining fundamental concepts like databases and graphs.

In Chapter 3 we present the auxiliary structures designed to search the databases. These structures are required for the matching process and to enhance the performance of the systems in terms of computational time. Generally, the systems proposed pre-compute indexes (similar to IR inverted indexes) mapping each term contained in the database with the locations of the term in the database schema or instance. The location-granularity and the details of the indexes are specific to each system and depend on the approach. Moreover, we surveyed the implementations of other structures designed to lighten the computational-heavy search algorithms.

In Chapter 4 we define the query as a set of keywords and present the query processing techniques exploited by the keyword search systems to enhance the performance and effectiveness of the retrieval. This techniques are adapted from the IR and natural language processing field, and include procedures like stemming and stopword removal. In this chapter we also address the *matching* process, where the database elements probably related to the query keywords are found. This process is usually realized using the indexes.

In Chapter 5, we present the different techniques and algorithms used to build the query answers. We survey the different structures proposed to connect the nodes containing the query keywords, and the way they are scored to output a ranked list. This kind of problems is generally hard in terms of algorithmic complexity and highly depends on the approach

followed by the authors. In this chapter we analyze how the systems evolved though the year, and outline the solutions that helped to improve the performance both in terms of effectiveness and efficiency.

Chapter 6 do not address a specific component of the pipeline, but presents the problem of *evaluate* the different systems. As for traditional IR, keyword search in databases provides results affected by a certain ambiguity and uncertainty, so that the evaluation in term of efficiency and effectiveness of these systems is fundamental to the progress of the field.

Finally, we draw the conclusion of this thesis in Chapter 7 , and provide some considerations about the future of this research area.

2 Background

2.1 The Relational Database Model

In this section the relational database (RDB) model is introduced. This term refers to a specific data model with relations as data structures and algebra for specifying queries. It was first formalized in [Codd, 1970] and later improved and expanded by researchers in database theory.

Among all the data model proposed in literature, the relational model is the most widely used, and the vast majority of current database management systems (DBMS) is based on it [Park and Lee, 2011].

An example of relational database is provided in Figure 2.1: it represents the data organization of a cinemas network show schedule. Intuitively, the data is represented in tables where each row contains data about a specific object or set of objects, and rows with uniform structure and intended meaning are grouped into tables [Abiteboul et al., 1995].

A table in the RDB model is called *relation*, and it is defined by its own

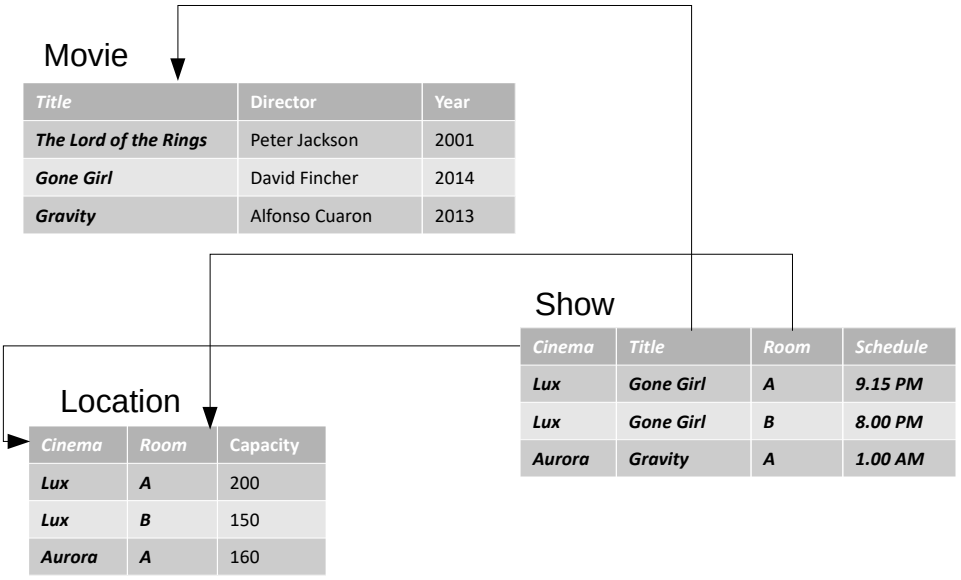

Figure 2.1. *Example of relational database* $\mathbf{R} = \{Movie, Show, Location\}$.

name. A relation row is called tuple, t, while a column is called *attribute*, a. Finally, U represents the set of all attributes of a relation.

A *relation schema* R is a relation name associated with its attributes: it is possible to define R as $R[U]$. The *arity* n of a relation R represents the number of attributes of a relation. In Figure 2.1 there are three relation schemas, *Movie[{Title, Director, Year}], Location[{Cinema, Room, Capacity}] and Show[{Cinema, Title, Room, Schedule}]*.

A *database schema* is a non-empty finite set **R** of relation names, that could be written $\mathbf{R} = \{R_1[U_1], ..., R_n[U_n]\}$ to indicate the relation schemas in **R**. In Figure 2.1 $\mathbf{R} = \{Movie, Location, Show\}$.

A *relation instance I = r(R)* of a relation $R[U]$ is a finite set of tuples with

arity $|U| = n$. A *database instance* of database schema **R** is a mapping **I** with domain **R**, such that $\mathbf{I}(R)$ is a relation over R for each $R \in \mathbf{R}$.

If t is a tuple of a relation instance I and $PK = (a_1, a_2, ...)$ is a set of attributes such that $t[PK]$ is the set of values of t, with reference to the attributes PK, then PK is a *primary key* if $\forall t_1, t_2 \in I, t_1[PK] \neq t_2[PK] \neq \emptyset$.

In simple terms, a *primary key* is a set of attribute values that define uniquely each tuple, so that two distinct tuples belonging to the same relation instance cannot share the same primary key. Primary key in Figure 2.1 are highlighted with bold font: In the *Movie* relation the *Title* attribute is sufficient to describe a primary key (we suppose that there aren't movie with the same title), while in Location and Shows this more attributes are needed.

A set of attributes $FK = (b_1, b_2, ...)$ of a relation $R_1[U_1]$ it's called *foreign key* with reference to relation $R_2[U_2]$ with primary key $PK = (b_1, ..., b_n)$ if $dom(a_i) = dom(b_i), 1 \leq i \leq n \wedge \forall t_1 \in r(R_1), t_1[FK] = NULL \vee \exists t_2 \in r(R_2) \mid t_1[FK] = t_2[PK]$. Then, a foreign key is a set of attributes of a table that refers to the primary key in another table, thus a foreign key cannot present values that do not appear in the other table's primary key. In Figure 2.1 foreign key constraint are represented through arrows pointing the referenced value.

2.2 Relational Database as a Graph

In many applications a relational database can be seen and materialized as a *graph*. Before addressing this particular case, the formal definition of graph is provided.

2.2.1 Graph Definition

An *undirected graph* is a pair $G = (V, E)$, where V is a finite set of *vertexes* and $E \subseteq V \times V$ is a set of *edges* connecting those vertexes. If $u, v \in V$ are vertexes of V and exists an edge between them, this *undirected edge* is defined as $(u, v) = (v, u)$.

In a *directed graph* G_d (also known as *digraph*) all the edges are directed, then $(u, v) \neq (v, u)$. It is always possible to transform an undirected graph into a directed graph replacing each undirected edge (u, v) with two opposite directed edges (u, v) and (v, u).

A *directed path* in G_d is a nonempty sequence $p_d = (v_0, ..., v_n)$ of vertexes such that $(v_i, v_{i+1}) \in E$ for each $i \in [0, n-1]$. n is the *length* of the path.

An *undirected path* in G is a nonempty sequence $p_u = (v_0, ..., v_n)$ of vertexes such that $(v_i, v_{i+1}) \in E \vee (v_{i+1}, v_i) \in E$ for each $i \in [0, n-1]$.

A *cycle* (directed or undirected) is a path $p = (v_0, ..., v_n)$ where $v_0 = v_n$. A graph is acyclic if it has no cycles.

Two vertexes $u, v \in V$ are connected if there is an undirected path in G from u to v, and they are *strongly connected* if there are directed paths from u to v and from v to u.

The *distance* $d(a, b)$ of two nodes a, b in a graph is the length of the shortest path connecting a to b [$d(a, b) = \inf$ if a is not connected to b]. The *diameter* of a graph G is the maximum finite distance between two nodes in G.

The *degree* $deg(v)$ of a node v is the number of edges incident to the vertex. The *in-degree* $deg^-(v)$ is the number of incoming edges to the vertex (*in-edges*), while the *out-degree* $deg^+(v)$ is the number of outgoing vertex (*out-edges*). It follows that $deg^-(v) + deg^+(v) = deg(v), \forall v \in V$.

A *tree* is a graph that has one and only one vertex with no in-edges, called the *root*, and no cycles. For each vertex v of a tree there is a unique proper path from the root to v. A *leaf* of a tree is a vertex with no out-edges. A *forest* is a disconnected graph that consists of a set of trees.

2.2.2 Materializing RDBs as Graphs

We formalize in this section two possible representation of a database as a graph. This passage is necessary to present the approaches proposed by the scientific community to respond to the keyword search in database problem.

A database schema could be represented as a graph $G_S(V_s, E_s)$ where $V_S = \{R_1, ..., R_n\}$ represents the set of schema relations of the database and E_s represents the set of edges between two relation schemas. $V(G_S)$ and $E(G_S)$ denote respectively the set of nodes and the set of edges of a graph G_S. Given two relation schemas, R_i and R_j, there exists an edge in the schema graph, from R_j to R_i, denoted $R_j \rightarrow R_i$, if the primary key defined on R_i is referenced by the foreign key defined on R_j. There may exist multiple edges from R_i to R_j in G_S if there are different foreign keys defined on R_j referencing the primary key defined on R_i. In such a case, $R_j \xrightarrow{X} R_i$ is used, where X is the foreign key attribute name.

Referring to instances of databases, an RDB can be viewed as data graph $G_D(V, E)$ on the schema graph G_S. Here, $V(G_D)$ represents a set of tuples, and $E(G_D)$ represents a set of edges between tuples. There is at least an edge between two tuples t_i and t_j in G_D, if there exists a foreign key reference

from t_i to t_j in the RDB. The number of edges and their direction between two tuples depends upon the requisite of the system that implement the data graph.

In general, two tuples, t_i and t_j are reachable if there is a sequence of connections between t_i and t_j in G_D. The distance $dist(t_i, t_j)$ between two tuples t_i and t_j is defined as the minimum number of connections between t_i and t_j.

2.3 Defining and Quering a Database

A database needs three theoretical different kind of language to be managed:

- A data definition language (DDL) must be defined to specify the database structure and dependencies, such as relations and foreign key constraints.
- A data manipulation language (DML) is necessary to add and modify data in the database.
- A data query language (DQL) allows the user to pose query to the database

To support this three different functions, SQL has been proposed. It is a standardized language that has established itself as the dominant language to relational database. Different versions of SQL expanding the functionality and the semantic of the standard one are implemented in commercial Relational Database Management Systems (RDBMSs) like DB2,

Oracle, MySQL or PostgreSQL. SQL was originally developed under the name Sequel at the IBM San Jose Research Laboratory.

The basic building block of SQL queries is the *select-from-where*. If the user wants to know the name of the director of the movie "Gone Girl" from the database shown in Figure 2.1, s/he have to provide to the system the following statement:

Listing 2.1. *An SQL simple query, posed on RDB in Figure 2.1, corresponding to the information need name of the director of the movie "Gone Girl"*

```
SELECT  Director
FROM    Movie
WHERE   Title='Gone␣Girl';
```

The DBMS will output all the cells with attribute name "Director" of each tuples of the table "Movie" where the value of "Title" cell is "Gone Girl". SQL language is powerful and complete, but unfortunately this completeness could be a drawback too, because to a non power user it is denied to query the database. In addition do this, posing a complex query to the database could be tricky even for an instructed user.

In the second example, referring again to Figure 2.1, we want to query the system to know how large are the rooms in which "Gone Girl" is going to play. This query require the user to identify all the tables storing relevant information, then know how to join them, and finally build a syntactically correct SQL query such as:

Listing 2.2. *A complex SQL query pose to the database in Figure 2.1, corresponding to the information need how large are the rooms in which "Gone Girl" is going to play*

```
SELECT Movie.Cinema, Location.Room, Location.Capacity
```

```
FROM   Movie INNER JOIN Show     ON  Movies.Title = Show.Title
             INNER JOIN Location ON  Shows.Cinema = Location.
                Cinema
                                 AND Shows.Room = Location.Room
WHERE  Movie.Title='Gone␣Girl';
```

To achieve this goal, the $JOIN$ operator ($\bowtie$ in relational algebra) has been used: it is a clause that allows to combine one or more tables from a database on one or more columns.

This operator is essential to keyword search schema-based approach, because it allows to build the output relations from the tables retrieved during matching process.

2.4 Keyword Search in Relation Database

Keyword search is the foremost approach for information searching and in the last decades it has been extensively studied in the field of Information Retrieval (IR) [Bergamaschi et al., 2013a]. It allows the user to pose extremely intuitive queries, intended as an unstructured sets (or ordered lists) of keywords defining her/his information need. The lack of specification of unstructured queries is balanced by the "best match" search paradigm, in which the system outputs a list of documents ordered by some similarity ranking function with respect to the query. The user is then invited to actively inspect the list, learn its content and possibly clarify her/his information needs by tuning the queries for finding better suited results.

Unfortunately, the keyword search model hardly adapts to structured

data sources like relational databases, which are typically accessed through structured queries as the above mentioned SQL. Structured queries are not user-oriented, because it requires the user to both manage the language syntax and know the structure of the data to be queried. Furthermore, structured queries are issued assuming that a correct specification of the user information need exists and that answers are perfect, i.e. they follow the "exact match" search paradigm.

An example clarifying the intrinsic difficulty of querying a database has been shown in Listing 2.2, while a simple keyword query $Q = \{Capacity\ Gone\ Girl\ rooms\}$ for the same information need (how large are the rooms in which "Gone Girl" is going to play) could have been more easily posed.

The stumbling block to overcome when adapting keyword search to databases is how to manage the differences between the information organizations in traditional IR and RDBs. While in IR the logical information unit is the *document*, there is not a equivalent concept in databases: information is scattered across several relations and tuples, and the possible interpretations of a keyword query correspond to the different ways by which their respective tuples are connected. To achieve this goal, several systems have been proposed in the last fifteen years, implementing a plethora of approach and algorithms.

Across the different works, we outlines a common pipeline, with recurring blocks composing it. This architecture is synthesized in Figure 2.2, and consists of the following steps: (a) data processing, (b) query processing, (c) element matching, and (d) answer structuring. It worth noticing that not all the systems put the same importance to each step,

sometimes collapsing two or more block in the same process, according on the focus of the work.

2.4.1 Systems Architecture

Data Processing The main purpose of this step is to explicitly define how information data is handled and what kind of auxiliary structures are needed to provide keyword search functionality. Information data is initially stored in relational database instances, managed by DBMSs. Generally, to be able to exploit them, data must be reorganized in the proper graph representation, as defined in Section 2.5.

Moreover, auxiliary data structures similar to IR inverted indexes could be implemented to provide an efficient way of matching keyword to database element in the matching step.

Query processing In this process, the system analyze the query in order to optimize the query for matching or provide a way to infer the intrinsic meaning of the query. The majority of systems proposed do not pay much attention to this component, or entrust it to the underlying RDBMS functionality. Its minor role in the systems architecture reflects distance of this systems to a practical implementation in commercial products.

Some authors, whose systems rely on semantic comprehension of the query, use disambiguation and other techniques to infer the meaning of the query and (a) better match keyword with the database elements [Bergamaschi et al., 2011a,b] and (b) increase the effectiveness of the systems [Demidova et al., 2010].

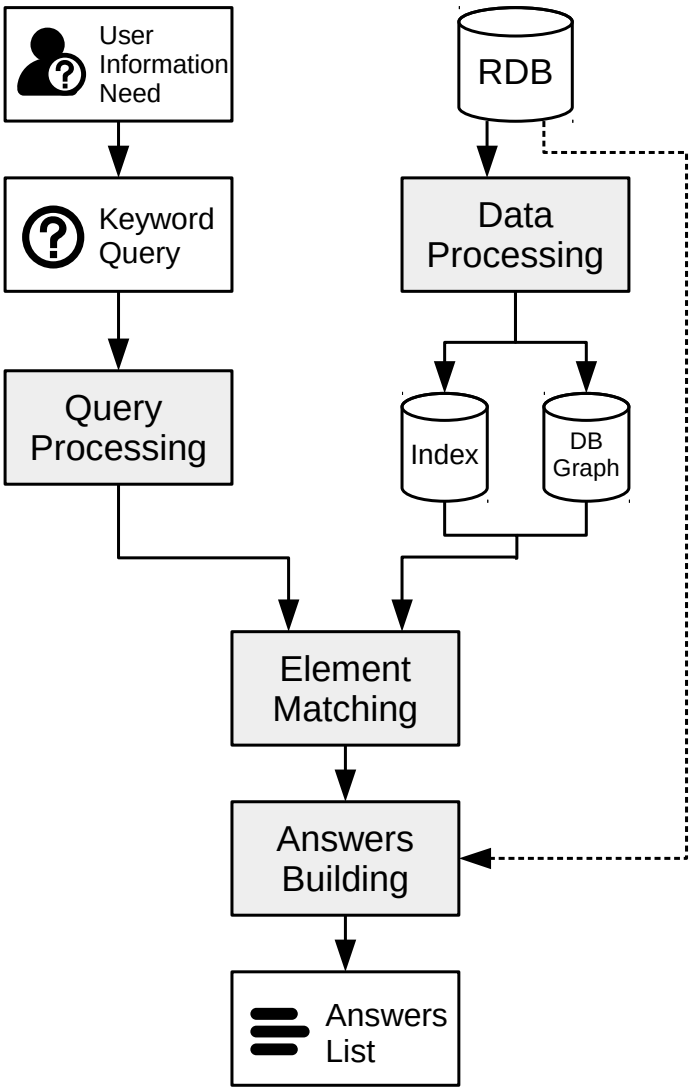

Figure 2.2. *A general keyword search in RDB system architecture.*

Matching This step address the problem of matching keywords with the appropriate database/graph element. To do it, the majority of systems proposed exploits one ore more indexes, with several differences between the various implementation. Generally it is a sort of inverted index mapping, for each term, the location of the term at a certain granularity, such as relations, columns or tuples. Some systems proposed do not exploit any auxiliary structure to achieve matching purpose, relying on semantic comprehension of the query and guessing the matching with the schema attributes.

Answer structuring and ranking In this step, the query answers are built, ranked and shown to the user. This step depends largely upon the system approach and the ranking functions chosen. Unlike IR, the answers typology could differ among the systems and approaches.

Ranking is crucial when searching in databases, both for effectiveness and efficiency reasons. While the effectiveness reason is trivial, the efficiency is related to the ability of output only the top-k answers, rather then compute all the answers.

2.5 Systems Approaches

In the literature, most of the designs proposed so far follows two main approaches: schema-based and graph-based. Both these approaches materialize the database as a *graph*, exploiting the *schema graph* G_S and *data graph* G_D representation respectively (as defined in Section 2.2.2). Another approach, less applied than the others, propose to build virtual

documents from the information contained in the database, so that an efficient IR-like retrieval process could be implemented. It worth noticing that all the systems acts as a middleware upon the RDBMS, materializing and managing the graphs and the relative auxiliary structures (if needed) without altering or modifying the underling RDBMS. In Table 2.1 we provide a list with the references to the principal systems proposed in this survey, which peculiarities are exposed in their relative chapters.

2.5.1 Schema-based Approach

Schema-based approach models the database schema as a graph G_S, so that nodes are mapped to database relations and edges to relationships such as foreign key-primary key dependencies.

The matching step generally exploits indexes to map keyword to relations, then the obtained tables are used in the answers generation process following this steps:

1. Building the schema of each possible answer (generally trees of tuples), taking as input the matched element with the query keywords and the graph G_S. If there are more than one answer for a query, this phase tries to find the schema for each one of them. Each schema is ranked according to a certain function in order to provide an ordered list of answers.

2. Possibly generate appropriate SQL queries for each answer, using patterns applied to the obtained schema. The SQL queries are then posed to the database in order to obtain final answer tables.

System	Reference
DBXplorer	[Agrawal et al., 2002]
DISCOVER	[Hristidis and Papakonstantinou, 2002]
DISCOVER II - efficient	[Hristidis et al., 2003]
PRECIS	[Simitsis et al., 2007]
EFFECTIVE	[Liu et al., 2006]
SPARK	[Luo et al., 2008]
LABRADOR	[Mesquita et al., 2007]
MeanKS	[Kargar et al., 2014]
POWER	[Qin et al., 2009]
SQAK	[Tata and Lohman, 2008]
KEYMANTIC/KEYRY	[Bergamaschi et al., 2011b,a]
BANKS	[Bhalotia et al., 2002]
BANKS II - bidirectional	[Kacholia et al., 2005]
BLINKS	[He et al., 2007]
Golenberg et al.	[Golenberg et al., 2008]
DPBF	[Ding et al., 2007]
EASE	[Li et al., 2008b]
STAR	[Kasneci et al., 2009]
PruneDP	[Li et al., 2016]
PACOKS	[Lin et al., 2016]
Dalvi et al.	[Dalvi et al., 2008]
EKSO	[Su and Widom, 2005]
SAINT	[Jianhua Feng et al., 2011]

Table 2.1. *A list containing the references to the main works presented in this survey.*

The schema-based approach has been first formalized in **DISCOVER** [Hristidis and Papakonstantinou, 2002] and its main structure has been adopted by later works. The authors model the relational database as an *undirected* schema graph G_S. To match keywords to the database elements, they exploits the underlying DBMS full-text index extension to build a *master index* and efficiently accomplish the task. They introduce the *Multi Total Join Networks of Tuples* as final answers to result to the user, which are trees interconnecting the relation tuples earlier matched to the query keywords. MTJNTs must be *total* and *minimal*, so that each network must contain all the query keywords (AND semantics) and do not contains any useless vertex.

To build the MTJNTs, the system implement two different steps

- In *candidate network generation*, each possible tuple sets network containing the query keywords (the *candicate network*) is built. This algorithm is based on a breadth-first traversal of the schema graph starting from the relations that contain the keywords.

- In *candidate network evaluation*, MTJNTs are built starting form candidate networks. In order to do it efficiently, a greedy algorithm that produces a near optimal execution plan has been proposed.

The candidate networks are built in order of increasing size; smaller networks, containing fewer joins, are preferred. Results are ranked based on the *number of joins* of the corresponding MTJNT, assuming that a smaller tree is more relevant than a larger one. Each MTJNT is mapped to an SQL statement that joins the tables as specified in the tree.

DBXplorer [Agrawal et al., 2002] provides an indexing structure, the *symbol table*, which maps the query keywords to the relational entities, without the need of any particular DBMS functionality. The symbol table could handle different kind of schema element granularity, like column level or cell level. DBXplorer results *join tree*, a simplified notion close to the candidate networks of DISCOVER. The answer building process is quite similar to that of DISCOVER, and both rank the tree with their number of joins.

In **DISCOVER II** [Hristidis et al., 2003] the authors implement for the first time state-of-the-art IR-knowledge to design the system. In continuation with DISCOVER work, they adapt their earlier algorithm for candidate network evaluation so that it results top-k MTJNTs for each candidate network, exploiting a monotonic IR-like ranking function, and avoiding to compute all the possible solutions. To do this, they proposed three different algorithms. Their *Sparse* algorithm is an enhanced version of the DISCOVER one that does not execute queries for non promising candidate networks. The *Global Pipelined* algorithm progressively evaluates a small prefix of each candidate network in order to retrieve only the top-k results, and hence, it is more efficient for queries with a relatively large number of results. The third algorithm is the *Hybrid*, that tries to estimate the expected number of results for a query and chooses the best algorithm accordingly.

In **PRECIS** [Simitsis et al., 2007] the authors propose a system that does not results a list of SQL queries, but generates a logical subset of the original database that contains not only items directly related to the given

query terms but also items implicitly related to them in various ways. In order to find the schema of the possible answers for a query, the logical subset creation algorithm first extracts initial subgraphs from the schema graph, each one corresponding to a different query interpretation. Then, these interpretations are enriched in order to discover other information that may be implicitly related to the query. This corresponds to expand the initial subgraphs based on a set of constraints. The creation of the initial subgraphs resembles the candidate network generation, but exploits a best-first transversal instead of the breadth-first-transversal. To generate the results, it uses an adaptation of the candidate evaluation algorithm of DISCOVER.

Effective [Liu et al., 2006] has been the first work to address the effectiveness issue instead of efficiency. They adapt the framework proposed in [Hristidis and Papakonstantinou, 2002] to generate tuple trees as answers for a given query, proposing a new ranking method that take into consideration the structure of each tree. In addition, they recognized two different kind of terms: one contained in text column, i.e. *value terms*, and the other *schema terms*, contained in tables, column and database name. They proposed a system to better rank queries that match schema terms, exploiting a synonyms table and introducing *schema-based document frequency*. Finally, they introduce the idea of *phrase*: they do not consider keywords only as single terms, but they use term position information in the columns (information stored in the inverted indexes) to infer which query terms must be considered part of a phrase.

In **SPARK** [Luo et al., 2008; Yi Luo et al., 2011] the authors propose a new non-monotonic ranking function to evaluate candidate network, modeling the join tree of tuple as dynamic, query dependent *virtual documents*, and ranking these documents with traditional-IR score. To provide top-k answers exploiting a non-monotonic function, they introduced three new retrieving algorithms, *Skyline Sweeping*, *Block Pipeline* and *Tree Pipeline* that try to minimize database probes using several novel score upper bounding functions.

In **LABRADOR** [Mesquita et al., 2007] the authors proposed a different approach than DISCOVER, based on a probabilistic model. The system tries to match the terms present in the initial unstructured query with the attributes in the underlying database schema, thus producing a set of candidate SQL queries. A Bayesian network model is deployed to calculate a score value that expresses the likelihood of each candidate SQL query corresponding to the original unstructured query. Therefore, only the top-k SQL queries would be considered to be processed. The SQL queries generated are as general as possible, and they can potentially retrieve large sets of results. To deal with this, the system ranks the query results by using a second Bayesian network model that evaluates the likelihood of a query result satisfying the original unstructured query.

Another different approach has been proposed with **KEYMANTIC** [Bergamaschi et al., 2011a]. The authors deviate from DISCOVER approach to address the problem of retrieving data without having a prior access to the database, so that no auxiliary structure

or inverted index could be built to do the matching between keywords and database terms. They propose a technique to translate keyword queries into SQL queries adapting the Munkres (Hungarian) algorithm [Bourgeois and Lassalle, 1971], known to solve the *assignment problem* in polynomial time. Having to provide the top-k best assignment, the algorithm is not stopped after providing the best mapping. Furthermore, for each generation, the weights used to do the assignment are dynamically updated in order to take into consideration interdependencies of the different assignments. The weights used to in the algorithm are computed taking in consideration the whole dependencies between the query terms, assuming that the meaning of each keyword is not independent from the meaning of the others.

KEYRY [Bergamaschi et al., 2011b] has an identical goal, i.e. implement keyword search without an a-priori access to the database, but exploits *Hidden Markov Model* for mapping user keywords into database terms. The use of a HMM allows to model two important aspects of the searching process: the order of the keywords in a query (this is represented by means of the HMM transition probabilities) and the probabilities to associates a keyword to different database terms (by means of the HMM emission probabilities).

Power [Qin et al., 2009] demonstrate the feasibility of implementing keyword search in databases without relying on an middleware solution. It fully exploits the RDBMS in order to match data and build SQL queries without any precomputing required, using only SQL to compute all the interconnected tuple structures. The authors provides three different kind of tuple structures as answers: (a) connected trees up to certain size (b)

sets of tuples reachable from a root tuple within a radius and (c) sets of multi-center subgraphs within a radius. To compute all the connected trees, it proposes an approach to prune tuples that do not participate in any resulting connected trees, followed by query processing over the reduced relations. To compute all multi-center subgraphs, it proposes a three-phase reduction approach to effectively prune tuples from relations followed by query processing over the reduced relations. Finally, it uses a similar mechanism to compute all the multicenter subgraphs to process sets of tuples that are reachable from a root tuple within a radius.

MeanKS [Kargar et al., 2014] has been designed to improve the efficiency of the search exploiting the user interaction with the system. In particular, it identifies the database entities that are potentially interesting to the user based on the query keywords the user provides, exploiting full-text DBMS capabilities, and allows the user to specify their interests through a user interface. This associate to each keywords a *role*, allowing to avoid the generation of unnecessary answers.

2.5.2 Graph-based Approach

The Graph-based approach models the database as a data graph G_D, in which nodes are mapped to tuples and edges to relationships between tuples, such as primary key dependencies. Under this representation model, an answer to a keyword query is represented by a set of connected subtrees of G_D containing all the keywords in its nodes. Building a tree that contains at least one node for each query keyword and with minimal cost means address the classical Steiner tree problem [Hwang and Richards, 1992], a

well-known NP-hard problem. Moreover, it must be noticed that the data graph is orders of magnitude larger than the database schema graph, than to execute a great number of joins to connect tuples containing the keywords could present scalability issues.

Except from the data graph building and updating, the underlying database (graph-based solutions act as middlewares upon the RDBMS) is never accessed to provide the final solutions, differently from schema-based in which the final tuples networks are mapped to SQL query possibly posed to DBMS.

BANKS [Bhalotia et al., 2002] has been the fist system to adopt graph-based approach: it materializes the data graph as a *directed weighted* graph G_D and performs the *Backward Expanding* search strategy to build the answers: it define paths starting from each vertex containing a query keyword, executing a Dijkstra's single source shortest path algorithm for each one of them. The idea is to find a common vertex from which a forward path exists to at least one keyword node in the query. Such paths will define a *rooted directed tree* with the common vertex as the root and the keyword nodes as leaves . Answers are ranked using a notion of proximity coupled with a notion of prestige of nodes based on in-links, similar to techniques developed for Web search (i.e. PageRank [Brin and Page, 1998]).

Due to the scalability issues of graph-based solution exposed above, Backward Expanding search performs poorly in case of a query keyword matching a very large number of tuple nodes. For these reasons, the same authors proposed in **BANKS II** [Kacholia et al., 2005] the *Bi-directional*

Expansion strategy, that improve Backward Expansion allowing forward search from potential roots towards leaves. Moreover, to avoid the BANKS bad performance in presence of high-degree nodes, the authors implemented an heuristic of spreading activation which prioritizes nodes with low degrees, and edges with low weights during the expansion of iterators. However, the performance of both systems remains affected by high-degree hubs.

BLINKS [He et al., 2007] is a bi-level indexing and query processing scheme for top-k keyword search on graphs. BLINKS follows a search strategy exploiting a bi-level index to prune and accelerate the search. The two level index is designed to reduce the potentially large index space, partitioning the data graph into blocks: The bi-level index stores summary information at the block level to initiate and guide search among blocks, and more detailed information for each block to accelerate search within blocks. This approach allows to improve the performance at the cost of some space occupation with respect to BANKS approach.

Unlike the approximate algorithms of BANKS and BLINKS, **DPBF**[Ding et al., 2007] propose a dynamic programming solution with a best-first strategy parameterized algorithm, able to find the optimal solution in reasonable time when the number of keywords in the query is very small. Its solution find the top-k optimal Group Steiner Tree Problem (and consequentially Steiner Tree problem) with time complexity $O(3^l n + 2^l((l + \log n)n+m))$ and space complexity $O(2^l \times n)$, where l is the number of groups

(in our case l i keyword length) , m is the number of graph nodes and n is the number of edges.

EASE [Li et al., 2008b] address the *r-radius Steiner graph problem*: known that graphs with a larger diameter are not so meaningful and relevant to queries the group Steiner tree problem, they propose to limit the size of the search space to an acceptable amount. A graph index materializes the information of the r-radius Steiner graph for maximal radius in advance to efficiently compute the cost of the answer graph. When necessary, a smaller radius graph can be reconstructed from the corresponding super-graphs. Structural relevance (distance between tuples) and IR-style ranking measures (TF-IDF) are used for ranking the answers. Other then structured data, this research covers unstructured data and semi-structured data as well.

[Kasneci et al., 2009] propose a approximation of the Steiner Tree problem, and adapt it to top-k queries. In order to build a first interconnecting tree, **STAR** relies on a similar strategy as BANKS, but, instead of running single source shortest path iterators from each node of V, STAR runs simple breadth-first-search iterators from each terminal. STAR may exploit taxonomic information (when available) to quickly build a first tree, by allowing the iterators to follow only taxonomic edges, i.e. edges labeled by taxonomic relations (e.g. *subClassOf*). This way, STAR can quickly find a taxonomic ancestor of all nodes from V. In a second phase, STAR aims at improving the current tree iteratively by replacing certain paths in the tree by new paths of lower weight from the underlying graph.

[Li et al., 2016] propose **PruneDP**, a progressive GST algorithm based on DPBF: the algorithm works in rounds, reporting a sub-optimal and feasible solution with smaller error guarantees in each round, until in the last round the optimal solution is obtained. To speed up the algorithm, they implemented an A^*-search strategy to select the most promising state to expand and the unpromising states to be pruned.

PACOKS [Lin et al., 2016] exploits a progressive ant-colony-optimization-based algorithm, for approximating the top-k Steiner trees problem, which achieves the best answer in a step-by-step manner, through the cooperation of large amounts of searches over time, instead of in an one-step manner by a single search. This way, the high costs for finding the best answer are shared among large amounts of searches, so that low cost and fast response time for a single search is achieved.

2.5.3 Virtual Document Approach

This approach differs from the others two, because its systems are designed to build virtual documents off line, so that the retrieval process could be efficiently done. This approach is characterized by large memory occupation and high efficiency, but the effectiveness from the user point of view has never been proved.

EKSO [Su and Widom, 2005] proposes a system that crawl the database in advance in order to provide *text-objects*, i.e. the structure connecting a tuple t with all the others tuples connected by primary/foreign key relationship, and *virtual documents*, i.e. the concatenations of the text-objects attributes. Each virtual document, computed off line on, represents

a meaningful and integral information unit, so that the retrieval process can be adapted from the IR field. In particular, they are indexed and retrieved using DB2's Net Search Extender.

A similar solutions is proposed **SAINT** [Jianhua Feng et al., 2011]. This system represents the database as a graph G_T, where nodes are *tuple units* and edges are relations among them. A tuple units is first introduced in [Li et al., 2008a] and follows the text-object definition of [Su and Widom, 2005]. Differently the solution proposed in [Jianhua Feng et al., 2011] allow to search on G_T, exploiting the connections between tuple units, outputting graphs of tuple units if the systems evaluate that the answers to the query resides in more than one tuple unit.

[Nandi and Jagadish, 2009] introduced the concept of **Qunits**, i.e. basic, independent semantic unit of information in a database. Qunits are not generated automatically, but must be built by someone who has a mental maps of the underlying databases. It is important to notice that the qunits building is then an subjective process, and this could eventually undermine the effectiveness of the search. For this reason, the authors proposed different possibilities to generate qunits, based on the database content, the keyword query history or on external sources.

3 Data processing

In information retrieval the practice of building auxiliary structures to improve the efficiency of search systems is well established. *The inverted index* [Zobel and Moffat, 2006], for example, has always been a fundamental and necessary element of nearly any information retrieval system since the foundations of this research area.

Generally, an IR-style inverted index is a table reporting, for each term of the document collection vocabulary, all the documents that contain the term. When posing a query to the system, this index can efficiently report the list of documents in which the keywords appear. Inverted indexes must be built before the user interaction with the system, thus it is necessary to have the possibility to access the underlying collection of document, and for each insertion or removal of document, the index must be updated.

When it comes to keyword search in relational databases, some data structures derived from the inverted index can be used to match keywords and database instance or schema elements, but unfortunately it is not possible to rank these matches and present them to the user in the

straightforward IR way. In IR, the logical information unit is represented by the *document*, so that the information need expressed through the query could be possibly fulfilled by the information contained in the document. This is not the case of databases: due to the normalization property, logical information units are scattered among different relations and tuples, so that it is necessary, after retrieving them, to connect them and build appropriate answers in order to fulfill the user information need. How these answers are built is the main argument of Chapter 5, whereas the main focus of this chapter is put on

- describing the auxiliary structures that allow to find the query-relevant elements of the database, both in schema- and graph-based systems

- provide a quick survey on the RDBMS functions that enable IR search on the underling databases allowing to build indexes on the relations columns, exploited by schema-based systems

In Chapter 2 the schema- and graph-based approaches have been discussed. The theoretical differences between these two approaches begin to materialize starting from the implementation of the graph in memory. Resuming, schema-based approach considers the database as a schema graph G_S, while the graph-based models it as a data graph G_D. This differences lead to the following considerations:

- Schema-based systems rely on a simple and lightweight schema graph without the need to update for each alteration on the database instance and without worrying for space occupation issue. They exploit

the database schema in order to connect matched elements during retrieval process.

- Graph-based systems need to build a potentially large graph connecting all the tuples of the database instance, say **I**. For efficiency reasons, the resulting graph must be resident in memory. Furthermore, the graph must be kept updated with **I**.

3.1 Indexing

As stated above, the task of an index is to efficiently match each query keywords with the database elements (both schema and instance) containing them. Indexes must be coherent with the approach used by the system in which it is implemented, thus the location granularity, namely the database location to which the keywords are mapped, may vary from an implementation to another. Using an index is a transversal practice among the approaches, therefore similarities can be found within schema-based and graph-based approach.

3.1.1 Schema-based Approaches

In schema-based systems, the authors present several different index designs, even if the solution proposed so far generally exploit the built-in full-text indexing and retrieval functionality of modern DBMSs.

DBXplorer [Agrawal et al., 2002] has been one of the first keyword search system to query databases. The authors implemented and compared

different *symbol tables* to establish their trade-offs. A symbol table is the equivalent of an IR inverted index, i.e. a map storing information about the location of words in the database at a given granularity, that can be tuple or column, or even cell. They analyze the different implementations evaluating three different factors: (a) space and time requirements, (b) effect on keyword search performance, and (c) ease of maintenance. The authors identified two interesting granularity levels: (a) column level granularity (Pub-Col), where for every keyword the symbol table maintains the list of all database columns that contain it, and (b) cell level granularity (Pub-Cell), where for every keyword the symbol table maintains the list of database cells that contain it.

Pub-Col symbol tables are usually much smaller than Pub-Cell symbol tables, because if a keyword occurs multiple times in a column (corresponding to different rows), no extra information needs to be recorded in Pub-Col. Moreover, Pub-col granularity requires less time when building the index.

Row level granularity has also been analysed, but it has been discouraged by the authors. It shows little advantage over the cell-level-one, while the relative subsequent answer building process is harder due to the absence of column information in the index.

The authors finally concluded that a column-level index must be preferred over cell-level because of its size, ease of update and performance, especially if database column indexes are exploited.

SQAK (SQL Aggregates with Keywords) [Tata and Lohman, 2008] exploits an inverted index based on column text to retrieve keyword-

data match. The authors choose Apache Lucene[1] to address the indexing task, specifically avoiding the use of DBMSs proprietary tools and easily implement SQAK over any commercial product.

In DISCOVER [Hristidis and Papakonstantinou, 2002] the authors explicitly exploited Oracle 9i Text extension to build their *master index*: first of all, they build an index on any database attribute, then all the indexes are sequentially inspected and combined to provide the master index. Given a keyword k_i, the master index task is to output the *basic tuple sets* $R_j^{k_i}$, $j = 1, ..., n$ where R_j is a relation of the schema, that are used in the answer structuring step. In [Hristidis et al., 2003] the master index is substituted by an IR index: while the building process remains the same, it stores useful information necessary to the system's new IR-style ranking function as tf, i.e. the *term frequency* of a word w in an attribute a_i, or df, i.e. the number of tuples in a_i's relation with word w in this attribute.

As for DISCOVER, many following systems [Hristidis and Papakonstantinou, 2002; Hristidis et al., 2003; Liu et al., 2006; Luo et al., 2007; Yi Luo et al., 2011; Simitsis et al., 2007] exploit indexes obtained through RDBMS feature. Major RDBMS vendors developed extensions that enable full-text indexing over text attributes in order to allow IR-style search on a single column. The management systems full-text search capabilities are implemented outside of SQL standardization, so that any vendor implemented them in a different way. For an overview of the main systems, refer to Section 3.2

[1] `http://lucene.apache.org/core/`

3.1.2 Graph-based Approaches

In graph-based systems the situation is less varied. BANKS and BANKS II [Bhalotia et al., 2002; Kacholia et al., 2005] exploit a simple inverted index that maps from keywords to table name/RowID pair, namely to a node of the graph. This approach is the only one adopted by all graph-based systems proposed.

The need for graph-based systems to keep the entire data graph G_D in memory could be seen as a limitation of this approach, but as stated in BANKS [Bhalotia et al., 2002] it is not unreasonable. The in-memory representation of the graph do not store any information about the relative tuples but only the *RowIDs*, while the index mapping keywords to RowIDs can be disk resident. In [Kacholia et al., 2005] the authors quantify the space occupation of a graph index in $16 \times |V| + 8 \times |E|$ bytes (basically a byte per node), than even large graphs with millions of nodes and edges could fit in tens of megabytes of central memory.

These theoretical assumptions contradict the systematic evaluation exposed in [Coffman and Weaver, 2014], where the graph-based systems hardly manage query posed to larger database because of space occupation problem (see Chapter 6

In [Dalvi et al., 2008] the possibility of graph with billions of vertex (i.e. the Web graph) that do not fit in memory has been addressed: the authors propose a *multi-granular* graph representation technique, which combines a condensed in-memory version of the graph with parts of the detailed graph

in in-memory cache. The multi-granular approach is based on the *2-stage* graph representation in which a Graph $G(V, E)$ is partitioned as following:

- The nodes, after clustering process, are grouped in *supernodes*, so that each supernode contains a subset of the innernode $v \in V$
- The *superedges* between the supernodes are constructed as follows: if there is at least one edge from an innernode of supernode s_1 to an innernode of supernode s_2, then there exists a superedge from s_1 to s_2.

The multi-granular solution enhance the 2-stage one allowing a supernode to be present either in *expanded* form, i.e., all its innernodes along with their adjacency lists are present in the cache, either in *unexpanded* form, i.e., its innernodes are not in the cache.

Since supernodes and innernodes coexist in the multigranular graph, several types of edges can be present, linking among them inner and super nodes. That solution avoids virtual memory approach, that potentially lead to poor performance due to the high number of I/O operations necessary for retrieving purpose.

3.1.3 Other

Virtual documents This particular approach propose to extract virtual documents from the databases that could be indexed and processed using traditional IR-approaches.

To achieve this, EKSO [Su and Widom, 2005] materialize the text-objects, i.e. the structure connecting a tuple t with all the others tuples connected by primary/foreign key relationship, and the *virtual documents*, i.e. the concatenations of the text-objects attributes. The indexing and

retrieving process are done on the virtual documents and entrusted to the DB2'2 Net Search Extender. This solution presents critical scalability issue, because experimental evaluation proved that indexes and structures of this approach could exceeded the size of the original database between two and eight times.

[Li et al., 2008a] introduced in literature the concept of tuple units, very close to the EKSO's text object, which definition is provided below.

Definition 3.1.1 (Tuple Unit)**.** *Given a database D with n connected relations, $R_1, R_2..., R_n$, for each tuple t_i in table R_i, let R_{t_i} denote the table with the same primary/foreign keys as R_i, having a single tuple t_i. The joined result of table R_{t_i} on other tables $R_j (i \neq j)$ based on foreign keys is called a tupleset. Given two tuple sets t_1 and t_2, if any tuple in t_2 is contained in t_1, we say that t_1 covers t_2. A tuple set is called a tuple unit if it is not covered by any tuple set.*

In [Jianhua Feng et al., 2011], the idea of tuple units have been expanded, designing a system able to find connections between tuple units. They proposed a graph G_T, where nodes are tuple units and edges are relationships between two tuple units. Given two tuple units, if they share the same value on any primary key attribute, they will be related, and thus connected. A term is said to be indirectly contained in a tuple unit t_i if there is a path between t_i and a tuple unit t_j that directly contains it. To allow an efficiently search, several structures are pre computed:

- The above mentioned graph G_T, with all the tuple units. This graph, with reference to G_D, is much smaller because it compacts group of tuples into the same node.

- A minimal distance matrix, containing the shortest paths from each node of the graph to the others
- A pivotal matrix, where each row is a tuple unit and each column is a keyword. In each cell, the matrix contains the tuple units that directly contains the term.
- A score matrix, where each row is a tuple unit and each column is a keyword, that contains a score computed taking into account IR score (tf-idf) and the minimum distance from the nodes containing the term and the other connected nodes.
- The SKSA or KPSA index.

The Single-Keyword-based Structure-Aware index is similar to traditional inverted index, but maintains both the directed and undirected tuple units that contain each term, with the relative scores. Differently, the entries of the Keyword-Pair-based Structure Aware index are keyword pairs $\langle k_i, k_j \rangle$ and contain a precomputed mutual score that would eventually be computed during the retrieval process. This final solution performs better in terms of speed, at the cost of more memory occupation.

LABRADOR In LABRADOR [Mesquita et al., 2007] the authors exploit Bayesian networks to infer the meaning of the user query, mapping each keyword to an attribute of the schema. To obtain this result, they use an inverted index that, for each term on the list, maps to a pair $\langle attribute\_id, frequency \rangle$ where *attribute_id* represents the table and column in which the term can be found, and *frequency* represents how many times the term occurs in the text values of the attribute.

No-Index Approach A different approach that avoids the use of indexes has been proposed in [Bergamaschi et al., 2011a,b]. They addressed the problem of not having an a-priori access to the database, so that no auxiliary structure nor index can be build. It is not a remote case: examples of such situations include databases on the hidden web and sources in data integration systems. The solution proposed exploits the inter-dependencies among the query keywords, assuming that the meaning of each keywords depends upon the meaning of the others, so that they collectively represent the information need that the user had in mind when posing the query. Than, using some auxiliary external knowledge, the system can infer the semantics that could represent those of the keyword query, mapping each keyword to schema terms, than output the best combination according to their ranking functions.

Conclusion

The variety of indexing implementations and auxiliary structures presented in this chapter are representative of the fragmented and various state of the current keyword search in database field. With such a scenario, to identify which implementation performs better is a difficult and critical task for the future.

It worth noticing that each system adapts the indexing and matching steps with regards to its own peculiarity, so that they cannot be treated as independent components but as integrated part of a complex system.

3.2 RDBMSs Full-Text Capabilities

Modern Relational Database Management Systems provide to the user some functions to query a database using keywords.

Even if the standard SQL operator LIKE is already implemented in all RDBMSs, it lacks in efficiency and capabilities, because it only allows the search in a defined column, scanning each tuple to highlight a hit without exploiting any precomputed result or index. Then, to implement IR-like search process on one or more column, the vendor introduced in their DBMSs query languge some new operators like CONTAINS or @@ (depending on the on the systems) to address the problem.

The new full-text extensions allows to first build an index on a set a column, than efficiently search among these columns using keywords. Moreover, these systems implement IR-style process like stemming, stoplist or query expansion to improve the effectiveness of the research. The various solutions proposed by the main RDBMSs commercial vendors are below exposed.

Oracle Text Oracle Text[2] is an extension to Oracle Database that provides specialized text indexes for traditional full text retrieval applications, and can filter and extract content from different document formats, like plain text, PDF or Word documents. Available from Oracle 9i, it is a rebrand of Oracle InterMedia extension.

Oracle Text provide three different type of index (CONTEXT, CTXCAT, CTXRULE) with different goals.

[2] `http://www.oracle.com/technetwork/testcontent/index-098492.html`

- Standard index type for traditional full-text retrieval over documents and web pages. The CONTEXT index type provides a rich set of text search capabilities for finding the content you need, without returning pages of spurious results.
- Catalog index type - the first text index designed specifically for eBusiness catalogs. The CTXCAT catalog index type provides flexible searching and sorting at web-speed.
- Classification index type for building classification or routing applications. The CTXRULE index type is created on a table of queries, where the queries define the classification or routing criteria.

To perform a keyword search on a database, the standard CONTEXT index is employed. To create an index, the following SQL statement must be provided (PARAMETERS clause specifies its optional features):

```
CREATE INDEX index_name ON Table(Column)
INDEXTYPE IS CTXSYS.CONTEXT
PARAMETERS (....);
```

Without specifying the parameters, the systems creates a default index which specification are described on the relative documentation. Oracle Text implements advanced *lexer* customization, based on the user preferences and the database peculiarities. It includes stopword removal feature, case insensitive capability and language specific functionality. To query such a CONTEXT index, the SQL clause CONTAINS is provided, allowing the system to output list of tuples ordered by the similarity SCORE computed between the keyword query and the text values on the indexed column.

```
SELECT SCORE(1), Column1
FROM   Table
WHERE  CONTAINS(Column1, 'k1␣k2␣k3', 1) > 0;
```

MySQL Full-Text Search MySQL Full-Text Search allows to create an index from the words of the enabled full-text search column and performs searches on this index. It is available from MySQL 5.6 with InnoDB or MyISAM storage engine[3].

MySQL supports indexing and re-indexing data automatically for a full-text search enabled column whose data type is CHAR, VARCHAR or TEXT. Defining a full-text index on an existing table is straightforward:

```
ALTER TABLE  Table
ADD FULLTEXT(Column1,Column2..);
```

or

```
CREATE FULLTEXT INDEX Table
ON Movies(Column1,Column2..):
```

To query the system, the MATCH-AGAINST function has been introduced. MySQl provides three different type of search:

- *Natural Language* is the default option: it allows to query the system in an easy and direct way.

- *Boolean Mode* introduces boolean operators $(+, -, <, >, (,), \sim, *, "")$ and allows to perform a search based on complex queries, suitable for experienced users.

[3] http://www.mysqltutorial.org/mysql-full-text-search.aspx

- *Query Expansion* is used to widen the search result of the full-text searches based on automatic relevance feedback. A search is then composed of three steps in which (a) the full-text search engine looks for all rows that match the search query, (b) it checks all rows in the search result and finds the relevant words, (c) it performs a search again but based on the relevant words instead of the original keywords provided by the users.

The following examples provide a general query in MySQL:

```
SELECT Column1, Column2
FROM Table
WHERE MATCH(Column1) AGAINST('k1␣k2␣k3'[mode selection]);
```

Unfortunately, MySQL full-text search is quite hostile to changes of the retrieving system: the different pipeline steps cannot be modify or fine-tuned by the user during run time, while some little variable changes could be applied at server startup time.

DB2 Text Search The IBM solution[4] supports plain text, HTML and XML formats.

The syntax to create an index is pretty similar to that of MySQL, except for the explicit format definition :

```
CREATE INDEX Schema.Index
FOR TEXT
ON Table(Column1, column2..):
```

The query syntax exploits the CONTAINS() function, in which Words or phrases can be combined with Boolean operators (AND, OR, NOT) and

[4] www.ibm.com/developerworks/data/tutorials/dm-0810shettar/

masked with wildcards (?, *) to limit or extend the search scope. The CONTAINS function results 1 if a tuple match the query. An example of DB2 keyword full-text query is provided below.

```
SELECT Column1, Column2
FROM Table
WHERE CONTAINS(Column1, 'k1␣k2␣k3') = 1
```

DB2 Text Search automatically uses stemmed forms in the search, along with others IR expedient .

SQL Server Full-Text Search From Microsoft SQL Server 2008[5] it is possible to use indexing and keyword query on a database columns. These columns can have any of the following data types: char, varchar, nchar, nvarchar, text, ntext, image, xml, or varbinary(max). Each full-text index indexes one or more columns from the table, and each column can use a specific natural language.

The queries allowed can search for any of the following:

- One or more specific words or phrases (simple term)
- A word or a phrase where the words begin with specified text (prefix term)
- Inflectional forms of a specific word (generation term)
- A word or phrase close to another word or phrase (proximity term)
- Synonymous forms of a specific word (thesaurus)
- Words or phrases using weighted values (weighted term)

[5] https://msdn.microsoft.com/en-us/library/ms142571.aspx

To achieve this, Microsoft SQL Server provide support for language-specific components like stemmers, stoplists, thesaurus files and filters, completely customizable by the database administrator.

The following statement is used to create and index with SQL Server, where in square brackets is to possible to adapt the different components:

```
CREATE FULLTEXT INDEX
ON table_name(column1 [...], column1 [...])
```

To pose the query, the systems provide two different predicates, CONTAINS and FREETEXT:

- CONTAINS must be used for precise or fuzzy matches to single words and phrases, the proximity of words within a certain distance of one another, or weighted matches. Some boolean operators are supported.
- FREETEXT must be used for matching the meaning, but not the exact wording, of specified words, phrases or sentences.

The final query is by now quite familiar (in this example the function FREETEXT has been preferred):

```
SELECT Column1, Column2
FROM Table
WHERE FREETEXT(Column1, 'k1␣k2␣k3');
```

PostgreSQL Full Text Search PostgreSQL supports since version 8.3 two kind of indexes, namely GiST and GIN, even if it is possible to text searching without prior indexing by sequentially fetch each tuples. The two kind of index are the following:

- Generalized Search Tree index - GiST - represents each document with a n-bit fixed length signature, hashing each word into a single bit in an n-bit string and than do OR operation together. This lossy algorithm may produce false match because the same signature could be generated by different documents, and this eventuality must be addressed by fetching the table records to resolve the false matches.

- Generalized Inverted Index - GIN - are not lossy but their performance depends logarithmically on the number of unique words.

Due to the fetching process, GiST indexes are generally three time slower than GIN ones, while the latter are two-to-three time larger than GiST indexes, and slower to build and update.

To build an index, simply define the table/column pair and the index typology:

```
CREATE INDEX index
ON Table
USING gist(Column);
```

PostegreSQL allows fine-grained control over how documents and queries are normalized, being able to have control over all the index preprocessing steps. In PostgreSQL the query is referred as *tsquery*, while a document, i.e. a text-value cell, as *tsvector*. The sintax to pose a query is quite different from the other solution proposed by competitors, and bit more complicated: the @@ operator is necessary to match a keyword query *tsquery* into a *tsvector*.

```
SELECT Column1, Column2
FROM Table
WHERE to_tsvector(Column1) @@ to_tsquery('k1 && k2');
```

4

Query Processing and Matching

Keyword search has become very popular to retrieve information on the web due to its efficiency and ease of use because the user do not have to be educated on a specific query language, but can express his/her information need with ease using natural language. We define a query as follows:

Definition 4.0.1 (Query). *A query $Q = \{k_1....k_n\}$ is defined as a set of keywords k_i given by the user and defining his information need.*

For a given query Q, the system must provide an answer or a list of possible answers aiming to fill the information need.

Aim of the keyword search system is to provide to the user a list of possible answers aiming to fill the information need expressed through a natural language query Q, overcoming the query inherently ambiguity. Although the definition of query is the same when searching in a document collection or a database, the output largely differs, because answers are built exploiting the structure of the database. Two kinds of possible queries are distinguished in [Guha et al., 2003]: *navigational* searches, where the user provides the search engine with a phrase or a combination of words which

he/she expects to find in the documents, and *research searches*, where the user provides the search engine with a phrase denoting an object he/she is trying to gather information about. Both IR and database keyword search systems address the first type of query.

Natural language queries are intrinsically ambiguous, thus each term could refer to multiple database elements. The process to mach each keyword to an appropriate element presents three critic aspects:

- The database content (schema or tuples) must be known in some way
- Each keyword must actively contribute to define the user information need
- The eventuality that multiple matches for the same query occur must be managed

The first point reflects the concepts exposed in Chapter 2 and Chapter 3 when the schema graph G_S and a data graph G_D database representations have been introduced, along with the auxiliary structures needed to accomplish the search task. Instead, the second and third aspects are the main focus of this chapter.

4.1 IR Query Analysis

In traditional information retrieval, a lot of work has been done to propose and improve query analysis techniques [van Rijsberben, 1979; Croft et al., 2009] to enhance both the efficiency and effectiveness of the search; whereas, when searching in a database, the community lacks of accuracy

and clarity when describing how the queries are handled. This behavior could have two different reasons: (a) the main focus of the papers in literature is put on the algorithms building up the final answers structures, (b) database researchers underestimate the role of the query processing components, accustomed to the rigid syntax of structured language.

Generally the systems proposed in literature exploit *exact matching* between the keyword and the database elements, thus the queries are assumed to be composed only by term denoting an element of interest to the user, without any stop-words or not essential terms such as redundant terms, conjunctions and propositions.

In a typical IR systems indexing pipeline, the query undergoes the same processing applied to any document in the collection, given that queries and documents are considered both homogeneous unstructured "bags of words". Generally, this pipeline include (a) tokenization, (b) stop-words removal, (c) stemming, (d) possibly some kind of terms expansion and (e) decompounding and phrases building. They can be seen as independent blocks of a chain, and can be implemented or not in any IR system.

Due to the structured nature of database information, as we have seen in Chapter 2 the implementation of these steps for both query and data is not straightforward as for traditional IR, but depends on the chosen keyword search approach and on system peculiarities.

In the following we analyze the IR pipeline from a database viewpoint, inspecting one by one the component listed above.

4.1.1 Stopwords Removal

Stopwords removal allows to discard any word that do not contribute to increase and determine the information content of a document or query. Stopwords are usually defined as functional words, like prepositions, pronouns, conjunctions, and articles, but are not limited to this classes. Generally *stoplists* are exploited to determine which words should be discarded. The relevance of a word in a collection is determined statistically: following the work of H.P. Luhn [van Rijsberben, 1979] the resolving power of significant words depends on the frequency of the term in the collection (or in the document); then, a term that appears few or too much times provide a little to none contribute to the document information.

In the majority of schema-based systems the RDBMSs full-text search capabilities are exploited to index the database columns and execute the matching (see Chapter 3). The stopword removal process is then entrusted to the database management system. Generally, as for Oracle or SQL Server, it is possible to define a custom stoplist and other implementation preferences, although any paper in literature exploit this possibility.

In graph-based systems stopword removal, as for the others query processing techniques, is never mentioned. In fact, the graph-based works mainly focus on the efficiency of the structuring process, without giving to much attention to the effectiveness issue.

4.1.2 Stemming

Stemming is the process where the variant word forms are mapped to their base form, namely the *stem* [Singh and Gupta, 2016]. After this

process, words like *write, writer, writing* could be substituted by their base form *writ*. Stemming is applied both to the query and the documents, allowing to have integrity between terms: effectiveness of the retrieval is this way increased, and considering the reduction of the dictionary's size, the efficiency is improved, too.

For systems that relies in RDBMs full-text functionality, the situation is the same as for the stopword removal in Section 4.1.1, thus the stemming process depends on the characteristics and setup of the underlying database system manager. Graph-based systems generally do not address the problem.

In [Agrawal et al., 2002] the authors conjecture the application of the stemming process on their system, along with other matching capabilities, such as synonyms or fuzzy matches, but do not implement any of them.

4.1.3 Query Expansion

As stated in [Furnas et al., 1987], the most critical language issue for retrieval effectiveness is the term mismatch problem, better known as *vocabulary problem*: the indexers and the users do often not use the same words to express the same concept. This problem manifests in two different aspects:

- Synonymy refers to different words with the same or similar meanings, such as 'tv' and 'television' (it decreases *recall*))
- Polysemy refers to the same word with different meanings, such as 'java' (it decreases *precision*)

One of the most usual techniques is to expand the original query with other words that best capture the actual user intent, or that produce a more useful query [Carpineto and Romano, 2012]: this is called *automatic query expansion* (or just query expansion) and has been developed to increase the effectiveness or IR systems since 1960s. Query expansion, is a category of techniques introduced to increase the number of possible matches between keywords and indexed elements, usually growing recall and decreasing precision.

To the best of our knowledge, query expansion has never been addressed directly in keyword search in databases, even though a similar approach can be found in [Liu et al., 2006], although not directly applied to the query terms.

In [Liu et al., 2006] the authors define two different kind of database terms: *schema terms*, i.e. the relation, attribute and domain names, and *value terms*, i.e. the words appearing in the columns. Schema terms usually do not occur in text values, so that potential relevant match are ignored. To avoid this, the schema terms (and not the query keyword) are expanded though external knowledge to include synonyms and increase the probability of relevant matching. A similar approach is proposed in SQAK [Tata and Lohman, 2008].

In the keyword search in relational database field, some researchers [Tata and Lohman, 2008; Zeng et al., 2016] have put effort on implementing aggregate queries (SQL queries containing aggregate functions like `sum`, `count`, `avg`, `max`, `min`...) in unstructured queries. The systems offering this possibility need to find a reference among the query

terms to the aggregate relative function and group-by clause. This matching is possible by searching for the exact term in the query or by exploiting a list of synonyms.

4.1.4 Segmentation and Phrases

Generally, when referring to a keyword, we mean one single word, useful to define the user information need. This definition could be relaxed introducing keywords made by several words. This kind of keywords are called *phrases*. For example, if the user wants to know who directed the movie "The Lord of the Rings" searching on the database provided in Figure 2.1 on page 8, a possible query posed to the system could be $Q = \{$"Director", "The Lord of the Rings"$\}$ where the movie name is treated as one single keyword. Even if research in IR has shown that phrase-based search can actually improve effectiveness [Liu et al., 2004], not all the systems proposed to search in databases can handle phrase-based query.

[Liu et al., 2006] proposed a system that in the inverted index stores, for each word, its position within each cell in which it appears: if the terms of a sub-query $P \subseteq Q$, $P = \{k_i, k_{i+1}, ..k_j\}$, where $i < j$, appear in column D, and k_{i-1} and k_{j+1}, if they exist and belong to Q, do not appear in an adjacent location to k_i and k_j respectively, then the system defines P as a phrase in D, and treats it like a normal keyword when computing ranking functions. Thus the system can automatically infer which keywords belong to the same phrase without the explicit help of the user. The authors designed the ranking functions to give more importance to a phrase keyword, with respect to simple keywords.

On the contrary to [Liu et al., 2006], SPARK [Luo et al., 2007, 2008; Yi Luo

et al., 2011] allows the user to indicate which terms belong to a phrase by explicitly quoting a phrase.

4.2 Query Language and Semantics

The first systems, both schema and graph based, proposed in the literature manage a limited query language to search the database. They generally implement conjunctive keyword semantics (boolean AND) and do not provide the user with wildcards and functional operators. For this reason these systems present to the user only structures containing all the query keywords. In the subsequent systems, query languages have been improved, allowing the user to pose more complex and complete queries.

Schema based systems [Hristidis et al., 2003] introduced the OR semantics in schema based systems, by allowing them to output tuple trees not necessary containing all the keywords.

SPARK system [Luo et al., 2007, 2008; Yi Luo et al., 2011] sets the OR semantics as default (same choice of [Liu et al., 2006], which authors state to allow a more flexible result ranking). Moreover, it provides the user with different operators like $-$ or $+$ to avoid or consider a keyword, quotes (‘ ’) to define phrase (see Section 4.1.4) or wildcards $*$ or . to define any character sequence with or without null sequence, respectively. [Simitsis et al., 2007] allows the user to explicitly use AND, OR, NOT operator to define the inclusion/exclusion of each single term in the query.

Graph-based systems To the best of our knowledge, all graph-based systems proposed so far are designed to support only boolean AND semantics, without any addition to the query language. It must be noticed that this is not due to any practical impedance but to the fact that their focus is put on the algorithm efficiency rather than on query expressiveness.

4.3 Matching

The matching process describes the steps required to find any possible match between a query keyword and a database element. In the systems which use inverted index-like structures, this process is straightforward and efficient: for each query keyword the index provides the list of the elements that contain it.

The systems not relying on indexes, such as [Bergamaschi et al., 2011a,b] exploit a different approach by trying to infer all the possible meanings of the query. This approach is described in the next section.

4.3.1 Matching Process Without Indexes

The main reason for not exploiting indexes for the matching process is to avoid the necessity of having an a-priori access to the database content. Possible scenarios where this could be helpful include the on-line access of databases through web interface, or the sources in information integration systems operating behind wrappers with specific query capabilities. This case has been addressed by Bergamaschi et al. in [Bergamaschi et al., 2011a,b] with two different approaches.

To do the matching, the systems must understand the intrinsic meaning

of the query and, as far as possible, manage the different misinterpretations. In order to accomplish this task, these systems see a query as an ordered list of keywords, instead of an unordered set, relying on the facts that the order of keywords is important and correlated keywords are typically close [Kumar and Tomkins, 2009]. Another crucial element is that the only knowledge about the database these systems need is the *schema graph* G_S. Eventually, the matching process leads to map each keyword with a *vocabulary term*, i.e. the set of all relation names, their domain names and their attribute names. These maps are called *configurations*.

Definition 4.3.1 (Configuration)**.** *A configuration C of a keyword query Q on a database* ***R*** *is an injective map from the keywords in Q to database terms in the vocabulary of* ***R****.*

There are three reasons behind the configuration injective propriety:

1. each keyword cannot have more than one meaning in the same configuration, i.e., it is mapped to only one database term

2. two keywords cannot be mapped to the same database term in a configuration, since overspecified queries are only a small fraction of the queries that are typically met in practice [Kumar and Tomkins, 2009]

3. every keyword is relevant to the database content, i.e., keywords always have a correspondent database term (query has been pre-filtered)

Based on this definition, the matching problem could be seen as finding the top-k configuration, with reference to specified weights and ranking

function. The two works proposed by Bergamaschi et al. differs on the process leading to define the various configurations.

In [Bergamaschi et al., 2010, 2011a] the authors proposed an approach where weights are computed to map each keyword to a schema element. In particular, they propose two different kind of weights:

- An *intrinsic* weight measures the likelihood of the fact that the semantics of the keyword is the same of the one of the database term, if considered in isolation from the mappings of all the other keywords in the query. To compute the intrinsic weights, the authors exploit techniques based on structural and lexical knowledge extracted from the data source, or based on external knowledge, e.g., ontologies, vocabularies, domain, etc.

- A *contextual* weight is used to measure the same likelihood but considering the mappings of the remaining query keywords. This is motivated by the fact that the assignment of a keyword to a database term may increase or decrease the likelihood that another keyword corresponds to a certain database term.

After computing these weights, the authors adapted the Hungarian (Munkres) algorithm [Bourgeois and Lassalle, 1971] to generate the best configurations, allowing the system to take into consideration interdependencies of different assignments.

Differently, in [Bergamaschi et al., 2011b] the authors model the matching function as a sequential process where the order is determined by the keyword ordering in the query. In each step of the process, a single

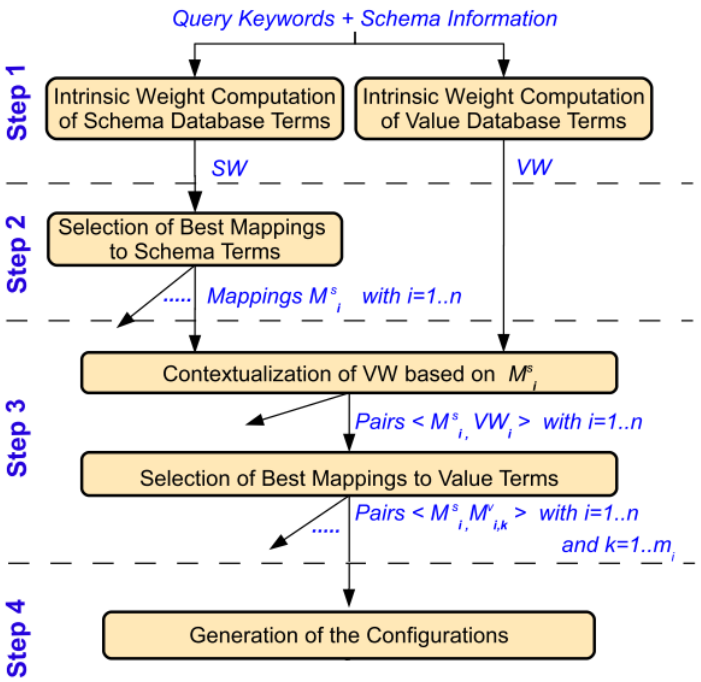

Figure 4.1. *Configurations building process from [Bergamaschi et al., 2011a]*

keyword is matched against a database term, taking into account the result of the previous keyword matches in the sequence. This process has a finite number of steps, equal to the query length, and it is stochastic, since the matching between a keyword and a database term is not deterministic. In fact, the same keyword can have different meanings in different queries and hence being matched with different database terms; vice-versa, different database terms may match the same keyword in different queries. This type of process can be modeled, effectively, using a *Hidden Markov Model* (HMM), which is a stochastic finite state machine where the states are hidden variables. The authors adapted a variation of the HITS algorithm [Li et al., 2002] for computing an authority score for each database term, to be considered as the initial state probabilities required by the HMM.

Conclusions

In this chapter we surveyed the different techniques used to process the query and match each term to an appropriate database element.

Concerning query processing, the argument has not been directly address in the literature. The main reason is that the attention of the researchers has been put on more critical topics like answers building, and generally the queries are then assumed to be *well-posed*. In any case, in most of the schema-based solutions, the RDBMS full-text functions are able to process the query through IR approaches.

The effectiveness of the keyword search in databases systems are presumably affected by the query processing pipeline adopted. Thus, in order to design better systems in terms of effectiveness from the user

viewpoint, we claim to a deeper research in this field, evaluating the different approaches following the guidelines proposed in Chapter 6.

Concerning the matching component, the inverted index approach performs well when applicable, while the database-ignorant approach by Bergamaschi et al. has never been compared to other systems in terms of performance and effectiveness. It is important to be aware of the space occupation issue of graph-based systems, presenting scalability problems when applied to large and complex databases. In sight of this, solutions like virtual memory usage or multi-granular graphs [Dalvi et al., 2008] have been proposed.

5 Answers Building

In the previous chapters the issues of matching the query keywords with the database elements, whatever information graph representation and approach we choose, have been addressed. In the keyword search in databases pipeline the subsequent step is to connect the matched elements to result structures that can fill the user information need. Two main factors affect this process:

- The approach chosen to materialize the graph and handle data, i.e. schema- or graph-based
- The semantics of the answer to result, which in turn depends upon the approach proposed

The structuring algorithms both for schema and graph based systems are outlined in the next sections.

5.1 Schema-based Approach

Schema-based systems perform searches on databases knowing only the database schemas G_S, while ignoring the database instances and lacking any information about the tuples that contain. The answer building process inputs the elements obtained by the previous matching step and uses this information to build relation networks from which SQL queries are obtained, exploiting predefined patterns. These SQL queries are finally posed to the DBMS in order to get the final view.

The schema-based approach originated with DISCOVER [Hristidis and Papakonstantinou, 2002], where the authors formalized the basic algorithms structure. The DISCOVER system defined the basis for most of successive schema-based systems. Contemporary to DISCOVER, DBXPLORER [Agrawal et al., 2002] expose similar concept with different notation.

The aim of these systems is to return *Minimal Total Joining Network of Tuples* (MTJNT), structures of tuples used to derive the SQL queries to retrieve data from the database.

Definition 5.1.1 (Joining network of tuples)**.** *A joining network of tuples J is a tree of tuples where for each pair of adjacent tuples $t_i, t_j \in J$ where $t_i \in R_i, t_j \in R_j$, there is an edge (R_i, r_j) in G_S and $(t_i \bowtie t_j) \in (R_i \bowtie R_j)$. The relation between t_i and t_j is than a parent/child relation.*

Definition 5.1.2 (Free tuple set)**.** *A free tuple set, denoted as $R^{\{\}}$, is simply a relation that appears in the schema graph, and that do not contain any keyword of the query.*

Definition 5.1.3 (Minimal Total Joining Network of Tuples)**.** *With reference to a query Q, J is a joining network of tuple with the following properties:*

- ***Totality*** *Every keyword $k_i \in Q$ is contained in at least one tuple of the joining network J*
- ***Minimal*** *If we remove one tuple from J, than $\bar{J}$ is no longer a total joining network of tuples.*

Definition 5.1.4 (Size on a MTJNT)**.** *The size T of a Minimal Total Joining Network of Tuples is the number of joins involved.*

A system design to search in databases must avoid the proliferation of redundant and useless MTJNTs. While the first adjective is self explanatory, to define what is useless is more complex. DISCOVER considers useless a *redundant* structure, i.e. a tree containing the same nodes and connections of a previously computed tree. Any non-redundant MTJNT is produced through *candidate network generation.*

Definition 5.1.5 (Candidate Network)**.** *Given a set of keyword $Q = \{k_1, ..., k_m\}$, a candidate network C is a joining network of tuple set, such that there is an instance I of the database that has a MTJNT $M \in C$, and no tuple $t \in M$ that maps to a free tuple sets $F \in C$ contains any keywords. A candidate network must satisfy the total and minimal conditions defined for MTJNTs.*

Specifically, a CN is a join expression that involves tuple sets plus perhaps additional database free relations $R^{\{\}}$. Intuitively, the free tuple sets in a CN do not have occurrences of the query keywords, but help connect via foreign-key joins the (non-free) tuple sets that do have non-zero scores for the query. The DBXplorer *join trees* could be seen as a simplified version of a the *candidate network* [Hristidis and Papakonstantinou, 2002].

Candidate Network Generator

The aim of this stage is to output a set of candidate networks, having a query $Q = \{k_1, .., k_m\}$ and a maximum network size T as input. This set of candidate network $\mathbf{C} = \{C_1, C_2, ...\}$ must follow this two properties:

- **Complete**: For each solution of the keyword query, it exists a candidate network $C_i \in C$ that can produce it.
- **Duplication-Free**: For every two CNs $C_i \in C$ and $C_j \in C$, C_i and C_j must have different structure.

To start the process, the system receives the output of the matching process described in Chapter 4.3 The keyword query Q and an index are exploited to retrieve a set of *basic tuple sets*, i.e.

Definition 5.1.6 (Basic tuple set)**.** *With reference to the keyword k_j, it is a set $\bar{R}_i^{k_j}$ for $i = 1, ..., n$ which consists of all the tuple relations R_i that contains the keyword k_j.*

After retrieving the basic tuple set, the *Tuple Set Post-Processor* uses them to produce the *tuple sets*:

Definition 5.1.7 (Tuple set)**.** *For every possible keyword subset $K_i \in Q$. a tuple set R_i^K consists in the tuples of R_i which contain all the keywords of the subset K and no other keywords, i.e., $R_i^K = \{t | t \in R_i \wedge \forall k \in K, t \text{ contains } k \wedge \forall k \in Q - K, t \text{ does not contain } k\}$.*

The tuple post-processor step has a computational cost that is exponential in the query size, but it is necessary for the systems to implement

AND semantic and then to take care of all the keywords [Hristidis and Papakonstantinou, 2002; Agrawal et al., 2002]. In DISCOVER II, [Hristidis et al., 2003] where an AND/OR semantics is implemented, it is not always necessary to compute and to list every tuple set for every relation-keyword subset combination. For this reason, the algorithm only create a single tuple set R^Q for each relation R. For queries with AND semantics, a post-processing step checks that only tuple trees containing all query keywords are returned. This expedient allows a faster and more efficient execution of the CN generator process.

The first algorithm designed to provide candidate networks is discussed in [Hristidis and Papakonstantinou, 2002]: it recursively expands tuple sets relations with adjacent relations (also empty free-tuple set $R^{\{\}}$), and discards networks according to the following pruning conditions [Yu et al., 2010]:

- Duplicated CNs are pruned
- A CN can be pruned if it contains all the keywords and there is a leaf node R_j^K where $K = \emptyset$, because it will generate results that do not satisfy the minimal condition
- A CN containing a subtree in the form R^K — S^L — R^M, where R and S are relations and the schema graph G_S has an edge $R \rightarrow S$, $K, L, M \subseteq Q$, must be pruned (the same tuple would appear in two different tuple sets, then the minimal condition would result violated).

The authors proved that the CNs generated with this algorithm are both total and minimal, namely that they could produce every possible MTJNTs and that this MTJNTs are not redundant. Unfortunately, the algorithm suffers from high computational cost and generates duplicate CNs that

have to be filtered out in a post-processing step. To avoid this problem, in [Markowetz et al., 2007] the *rightmost algorithm* was proposed: it makes pruning rules unnecessary by assigning a proper expansion order to the partial trees.

Candidate Network Evaluation

The candidate networks retrieved during the previous step must be evaluated to provide the final MTJNTs to the user. To achieve this, in DISCOVER, the *plan generator* module inputs a set of candidate networks and creates an near-optimal execution plan through a naive greedy algorithm, designed following two observations: subexpressions that are shared by most CNs and subexpressions that may generate the smallest number of results should be evaluated first. It must be noticed that the problem of finding the intermediate results to build, so that the overall cost of building these results and evaluating the candidate networks is minimum, is NP-complete, as stated in in [Hristidis and Papakonstantinou, 2002]. In DISCOVER an DBXplorer, the rank of a MTJNT is computed counting its number of joins, based the assumption that a larger network is less informative than a smaller one.

Successive systems tried to mitigate the intrinsic complexity of the problem avoiding the calculation of all possible results for a specific query, and only the top-k most scored results are computed. This approach is corroborated by the assumption that the user has a low interested on lower scored results. To achieve this goal, the algorithms must find a proper order of generating MTJNTs in order to stop after k results.

DISCOVER II [Hristidis et al., 2003] proposed three different algorithms

to solve the problem. All the proposals use an attribute level ranking function, computed using IR-derived metrics such as term frequency tf (of a term in an attribute) or *document frequency df* (the number of tuples in a_i's relation containing the term). The algorithms relies upon the *tuple monotonicity* property, necessary to stop the algorithm as soon as possible. The effectiveness problem has been later addressed in [Liu et al., 2006], where the authors proposed a new ranking function that takes in consideration both the morphology and the IR component of the tuple trees.

Definition 5.1.8 (Tuple Monotonicity)**.** *The property imposes that for any two MTJNTs* $T = t_1 \bowtie t_2 \bowtie ... \bowtie t_l$ *and* $T' = t'_1 \bowtie t'_2 \bowtie ... \bowtie t'_l$ *generated from the same candidate network, if for any* $1 \leq i \leq l$ $score(t_i, Q) \leq score(t'_i, Q)$*, then* $score(T_i, Q) \leq score(T'_i, Q)$*.*

The three algorithm proposed in [Hristidis et al., 2003] are:

- The *Sparse* algorithm, which computes a bound MPS_i on the maximum possible score of a tuple tree derived from a CN C_i. As a further optimization, the CNs for a query are evaluated in ascending size order. This way, the smallest CNs, which are the less expensive to process and are the most likely to produce high-score tuple trees using the combining function above, are evaluated first.

- The *Single-Pipelined* algorithm, which receives as input a candidate network C and the non-free tuple sets TS_1,..., TS_v that participate in C. The algorithm keeps track of the prefix $S(TS_i)$ that it retrieved from every tuple set TS_i; in each iteration, it retrieves a new tuple t from a TS_M, after which it is added to the associated retrieved prefix $S(TS_M)$. Then, the algorithm identifies each potential joining tree of tuples T in

which t can participate. The algorithm retrieves all joining trees of tuples that include t and adds them to a queue R. To empty R, it is necessary to guarantee that they are one of the top-k joining trees for the original query. In order to do this, a bound of the score achieved so far is mantained.

- The *Global-Pipelined* is the most efficient algorithm proposed by the authors. All CNs of the keyword query are evaluated concurrently following an adaptation of a priority preemptive, round robin protocol, where the execution of each CN corresponds to a process. Each CN is evaluated using a modification of the Single Pipelined algorithm, with the "priority" of a process being the bound value of its associated CN.

MeanKS In order to facilitate the computation of the results, in [Kargar et al., 2015] the authors proposed to the user, after the matching process, to define a specific role for each keyword among various possibilities provided. The *Minimal Joining Networks of Tuples Covering Roles* share the same definition as the MTJNTs, plus the *role and keyword covering* requirements, which force that for any query keyword role r_i (r_i is a relation in the database) it exists a node t_j in the tree T such that $t_j \in r_i$ and t_j contains keyword k_i. The user select the role filling a form similar to the one shown in figure 5.1

To rank the trees, the authors implement three different, mutually exclusive approaches based on the importance of the edges, of the nodes or both nodes and edges. They do not use any IR-based knowledge to rank them.

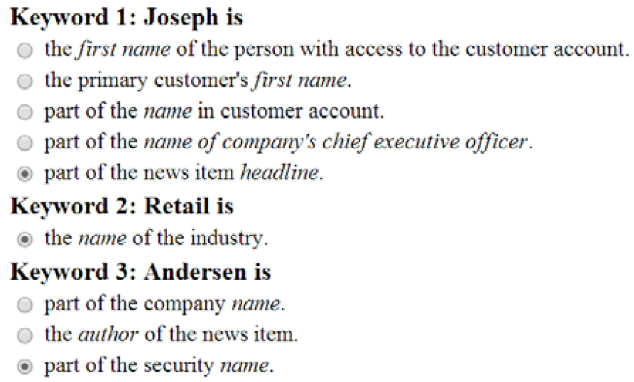

Figure 5.1. *Role selection by the user for query "Joseph Retail Andersen". (from [Kargar et al., 2015])*

SPARK In [Luo et al., 2008] the authors propose a tree level ranking function which does not satisfy tuple monotonicity. Basically, the authors model the tuple trees as virtual, query-specific documents, then assign an IR-like ranking score to the documents. Two new algorithms are proposed to solve the problem, based on the single-global pipeline algorithms of DISCOVER II: *Skyline Sweeping* and *Block-Pipelined*.

Précis Taking a keywords query as input, the aim of Précis [Simitsis et al., 2007] is to generate an entire multi-relation database instead of the typical individual relation that is outputted by other approaches. This database is a logical subset of the original one, i.e., it contains not only items directly related to the given query terms but also items implicitly related to them in various ways.

The logical subset schema creation process largely differs form the DISCOVER procedure, due to the different nature of their task, and is decomposed into two subproblems: initial subgraph creation and expansion. Giving as inputs the query and the schema graph G_S, the

initial subgraph creation process provides the most significant subgraph, according to the boolean query semantics, relative to a weighting function that takes in account the dimension and cost of the subgraphs, which nodes are weighted from their in-degree value. The second step aims to expand each initial subgraph adding a new edge, provided that the target relation is significant for all initial relations. The most valued graphs according to the same ranking function above are then output by the system.

QUEST The QUEry generator for STructured sources [Bergamaschi et al., 2013b] a search engine for relational databases that combines semantic and machine learning techniques for transforming keyword queries into meaningful SQL queries. The system relies on the Hidden Markov Model approach proposed by the same authors in [Bergamaschi et al., 2011b] to do the matching process, and exploits a hybrid approach to build the answers. Focusing on the structuring problem, the algorithm adopts a solution similar to the graph-based one. It materializes the database as a weighted graph G_A having each attribute of the database for nodes and edges connecting (a) the node representing the primary key of each table with all the other attributes in the same table, and (b) nodes associated with couples of primary-foreign keys. With such a graph, the algorithm aims to obtain the top-k Steiner trees using a mutual information-based distance to compute the weights of the edges, similar to what proposed in [Yang et al., 2011]. It must be noticed that the *configurations* (see Chapter 4.3) which define the nodes composing the Steiner trees do not assure that the trees are consistent with the database content and the user keywords, because the matching process map keywords into database terms in isolation, and

could lead to void tuples sets.

5.2 Graph-based Approach

This branch of systems represents the database as weighted data graph $G_D(V, E) \quad (where \quad u, v \in V, \quad e \in E)$, using graph algorithms to do the search. Weights are applied to nodes and edges according to the specific algorithm. Generally, for each direct edge (u, v) there is a backward edge (v, u) with a different weight. Usually, weights are computed according to a prestige notion, in a PageRank fashion [Brin and Page, 1998]

These systems could return two kinds of answer structures to the user:

- *Tree-based semantics* systems return trees containing all the keywords in their nodes
- *subgraph-based semantics* systems return subgraphs $S_i \subset G_D$. One particular case of these subgraphs is the *r-radius Steiner graph*, which definition is based on *centric distance* and *radius* as defined Section 5.2.

Among the above mentioned structure, the most popular one in literature is the tree-based. These trees are ranked according to two different semantics:

- In *Steiner tree-based* semantics the weight of a tree is defined as the total weight of the edges in the tree.
- In *distinct root-based semantics* the weight of a tree is the sum of the shortest distance from the root to each keyword node

Apart from the ranking function, the two semantics differ from the number of answers presented to the user: while the first allows the system to output up to $O(2^m)$ results ($m = |E|$), the second is limited to $n = |V|$, because each tree must be rooted on a different $v \in V$.

Tree-based Semantics

BANKS I-II The first graph-based systems conceived was BANKS [Bhalotia et al., 2002]. The authors proposed an algorithm implementing the Steiner tree-based semantics called *Backward Search*: it starts searching for connections from the keyword-nodes (the tuples that contains one or more keyword), following backward edges. This is accomplished by running as many copies of Dijkstra's single source shortest path algorithms as the number of matched tuples. The idea of concurrent backward search is to find a common node from which there exists a shortest path to at least one keyword-node for any keyword. Such paths will define a rooted directed tree with the common node as the root and the corresponding keyword nodes as the leaves. The trees are computed in increasing height order. It has been demonstrated that the backward search provides an l-approximation of the Steiner Tree problem, where l is number of query keywords.

To rank each tree, the following specification has been used:

- Forward edges have a default weight $w_{(u,v)} = 1$
- Backward edge weights are computes as $w_{(v,u)} = \log_2(i + indegree(v))$
- A score $s(T, t_i)$ for an answer tree T with respect to keyword t_i is

[1] `http://dblp.uni-trier.de`

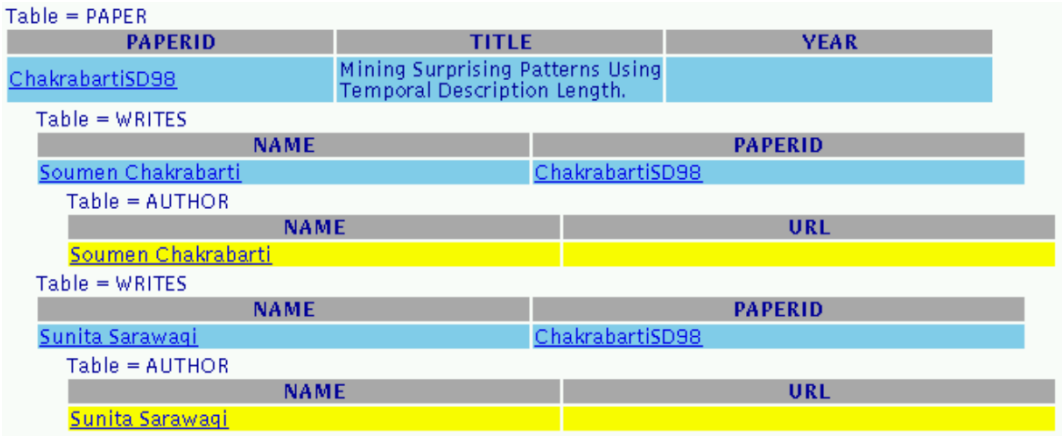

Figure 5.2. *Tree result of the query "soumen sunita" on DBLP*[1] *dataset using BANKS. (Taken form [Bhalotia et al., 2002])*

defined as the sum of the edge weights on the path from the root of T to the leaf containing keyword t_i.

- The aggregate edge-score E of an answer tree T is $\sum_i s(T, y_i)$
- The prestige of each node is determined using a biased version of the Pagerank [Brin and Page, 1998] random walk, where the probability of following an edge is inversely proportional to its edge weight taken from the data graph
- The tree node prestige N is the sum of the node prestiges of the leaf nodes and the answer root.
- The overall tree score is EN^{λ} where λ helps adjust the importance of edge and node scores (the authors claim to obtain the best result with $\lambda = 0.2$ in terms of effectiveness).

To speed up the retrieval process, the same authors proposed in BANKS II [Kacholia et al., 2005] the bidirectional search. One of the main drawback

of BANKS is that it instantiate a Dijkstra iterator for each keyword-node. With complex schema and many keywords in the query, this strategy results inefficient, because an iterator may need to explore a large number of nodes if it hits a node with high in-degree. The new algorithm aim to enhance backward search enabling a forward search starting from some potential root nodes found along the iterations. The BANKS II enhancements over BANKS are synthesized in three points:

- An *incoming iterator* is instantiated to follow the backward edges, doing the task done by the plethora of iterators in BANKS. It must be noticed that this is not a Dijkstra iterator, but the nodes to visited are decided following the *spreading activation* (see below)
- A *outgoing iterator* is instantiated to follow the forwarding edges starting from all the nodes explored by the incoming iterator
- They use *spreading activation* to prioritize the search. For the incoming iterator, the next node to be expanded is the one with the highest activation, a kind of "scent" spread from keyword nodes. The spreading activation is crucial also for deciding which iterator among the two to chose.

BLINKS In BLINKS [He et al., 2007] the authors propose a bi-level index search strategy. The purpose of the index is to precompute all the distances from the nodes to keywords in an efficient way, in order to enable a faster search on the data graph. BLINKS partitions a data graph into multiple subgraphs, or *blocks*. A bi-level index consists of a top-level block index, which stores the mapping between keywords and nodes to blocks, and an

intra-block index for each block, which stores more detailed information within a block. The authors studied the possibility of exploiting a single-level index to accomplish the task, but bumped into memory occupation issue. The bi-level index is then a compromise between performance and memory occupation needs. The retrieval algorithm is based on *cost-balanced expansion*, a new policy for the backward search strategy of BANKS I-II. It aims to expand the cluster with the smallest cardinality: once the cluster has been chosen, it includes the node with the shortest distance to this cluster origin (i.e. the initial matched tuple). The distance can be found in the earlier built index. The BLINKS scoring function $S(T)$ of an answer T follows the distinct root-based semantics, and is defined as $S(T) = S_r + S_n + S_p$, i.e. the sum of the scores relatives to the root, the leaves and the paths from the answer root to the leaves. The component score functions incorporate measures based on both graph structure (e.g., node scores reflecting PageRank and edge distances reflecting connection strengths) and content (e.g., IR-style TF/IDF scores for matches).

STAR STAR [Kasneci et al., 2009] proposed a similar strategy to BANKS for building and ranking answer trees based on Steiner tree semantics. They introduced an algorithm yielding to $O(\log n)$-approximated Steiner trees, that can be adapted to provide a top-k search in the graph. It runs a breadth-first-search iterators from each keyword-node in a round-robin manner, to build a first tree T. The main peculiarity of STAR is the possibility of exploiting node and edge labels applied on the graph to quickly to construct the tree, assuming that such information lead to a common ancestor of the visited nodes.

In the second phase, STAR aims to iteratively improve the tree T by replacing certain paths with new paths of lower weight from the underlying graph. To explain the process, we must first introduce the *Steiner nodes* of T , i.e. the non-terminal nodes (which does not contain any query keywords), the *fixed nodes* of T, i.e. the terminal node or the Steiner nodes with $deg(v) \geq 3$, and finally the *loose path*, i.e. path with minimal length in T, which end nodes are fixed nodes. According to these definitions, a minimal Steiner tree with respect to the query Q is a tree in which all loose paths represent shortest paths between fixed nodes. In each iteration, removing a loose path, the tree T is split in two subtrees T_1 and T_2, which must be connected through a new path with lower weight, as in Figure 5.3. The iteration continues until no other loose path can be replaced in T.

Dynamic Programming Best-First Algorithm In [Ding et al., 2007] the Steiner tree problem (or, better, the group Steiner tree variants) has been addressed implementing a dynamic programming solution with a best-first strategy, adapted to find the top-k answers of a keyword search in databases problem. The first assumption that exposes the substructure property of the problem is that each keyword-node v, that contains the keywords set $q \subseteq Q$, can be seen as a single node tree, rooted at v, with a cost zero (the weights are applied only to edges), i.e. $T(v, q) = 0$. The substructure property is defined by the following equations:

$$T(v, q) = \min(T_g(v, q), T_m(v, q))$$

$$where \quad T_g(v, q) = \min_{u \in N(v)} \{(v, u) \oplus T(u, q)\}$$

$$T_m(v, q_1 \cup q_2) = \min_{q_1 \cap q_2 = \emptyset} \{T(v, q_1) \oplus T(v, q_2)\}$$

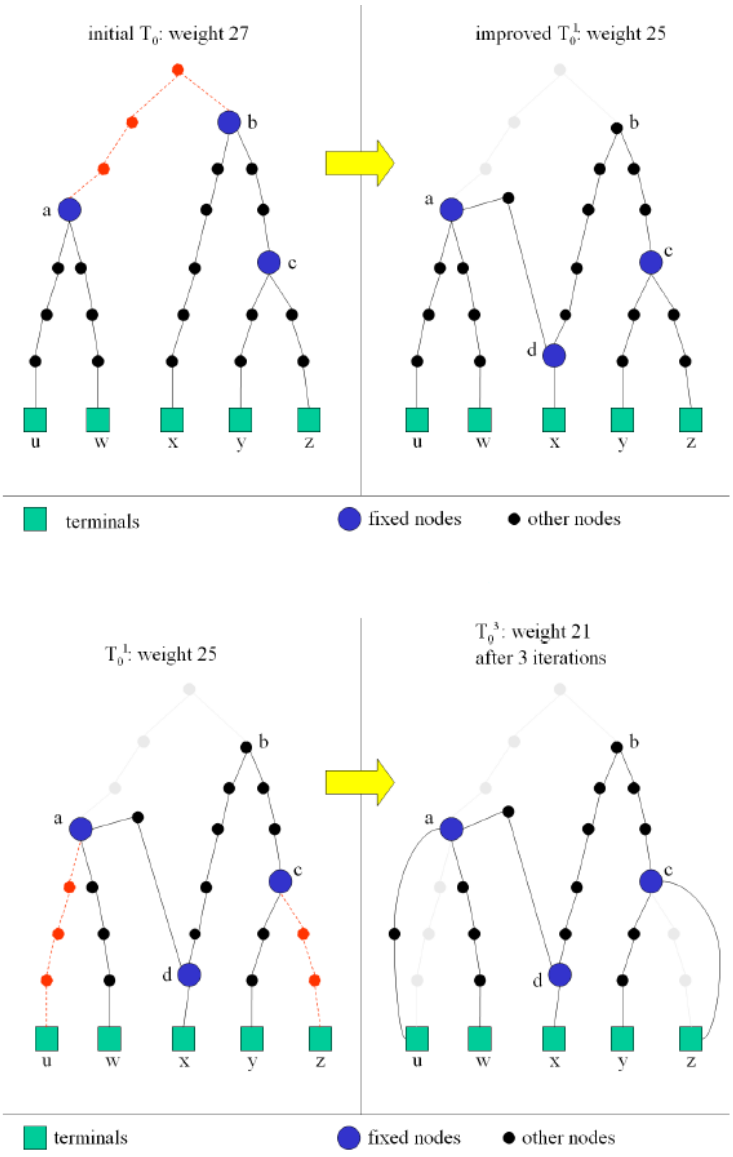

Figure 5.3. *Two iterations of the STAR algorithm to build a $O(\log n)$-approximated Steiner tree (from [Kasneci et al., 2009])*

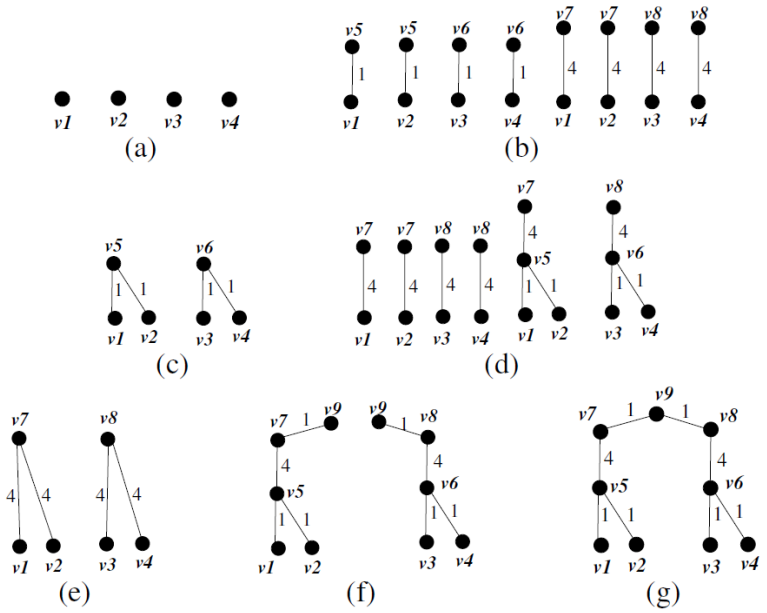

Figure 5.4. *A Best-First DP Solution (from [Ding et al., 2007])*

Here $\oplus$ is an operation to merge two trees into a new tree, and $N(v)$ is a set of neighbors of v such as $N(v) = \{u|(v,u) \in E(G_D)\}$ in the data graph G_D. An example of trees merging into an optimal Steiner tree can be seen in Figure 5.4. DPBF maintains trees in a priority queue, by the increasing order of costs of trees. The smallest cost tree is maintained at the top of the queue. The algorithm dequeues the top tree $T(v,q)$ of the queue and grows it exploiting the equations above, then the algorithm enqueues it and reorder the queue. If $T(v,q)$ contains the entire set of keywords Q, the algorithm will return $T(v,q)$ as the optimal Steiner tree and terminate. To implement the top-k solution, it is sufficient to not terminate after the first tree retrieved, but continue the iteration until the k-th.

Subgraph-Based Semantics

EASE [Li et al., 2008b] is an adaptive keyword search method for indexing and querying large collections of heterogeneous data, modeling unstructured, semi-structured and structured data as graphs, with weighted nodes as documents, elements and tuples respectively, and weighted edges as hyperlinks, parent-child relationships and primary foreign-key relationships respectively. The authors do not address the Steiner tree problem, but the r-radius Stainer graph problem, i.e. they aim to find all the graphs that contains all or a part of the keywords, and that have an acceptable size. The following definitions could help to better understand the problem.

Definition 5.2.1 (Centic Distance)**.** *Given graph G_D and any node v in G_D, the centric distance of v, denoted as $CD(v)$, is the maximal value among the distances between v and any node u in G_D, i.e., $CD(v) = \max_{u \in G_D}\{dist(v,u)\}$.*

Definition 5.2.2 (Radius)**.** *The radius of a graph G_D,denoted as $R(G_D)$, is the minimal value among the centric distances of every node in G_D, i.e., $R(G_D)$=$\min_{v \in G_D}\{CD(v)\}$. G_D is called an r-radius graph if the radius of G_D is exactly r.*

Definition 5.2.3 (r-Radius Steiner Graph)**.** *Given an r-radius graph G_D and a keyword query Q. A node in G_D is called a content node if it directly contains some input keywords in Q. A node s in G_D is called a Steiner node if there exist two content nodes, u and v, and s is on the path $\langle u, v \langle$ (s may be u or v).The subgraph of G_D composed of the Steiner nodes and associated edges is called an r-radius Steiner graph. The radius of an r-radius Steiner graph may be smaller than r but cannot be larger than r.*

To compute the r-radius Steiner tree, the systems precompute, using an adjacency matrix, each r-radius graphs of the dataset, one for each node $v \in G_D$. These graphs are query independent, and could be stored in secondary memory. Then, the structuring algorithm prunes the r-radius graphs in order to obtain r-radius Steiner graphs. The ranking functions used in EASE reflect its heterogeneous nature: it takes into account both document relevancy from the IR perspective and structural compactness from the DB perspective to capture structural relationships. To efficiently identify the top-k answers with the highest scores, the systems maintains a a sort of inverted index of all the query independent scores computed on the r-radius graphs, so that, before computing the Steiner graphs, the system retrieves the maximal r-radius graphs for the specific query.

PACOKS A progressive ant-colony-optimization-based keyword search algorithm is the argument of [Lin et al., 2016]. This algorithm aim to reduce the response time for a single search through the cooperation of large amounts of searches over time. To work, it exploit a single-step ant-colony-optimization-based algorithm for approximating the top-k Steiner trees problem (ACOKS). This component results a solution (or top-k solutions) closer to the global optimal solution at each iteration, but it needs a large amount of ants to get the optimal solution, unacceptably slowing down the performance. PACOKS algorithm aims to reduce this issue implementing ACOKS with a limited number of ants for each search, and relying on the continue use of the search engine. In other words, the result of the current search is a further optimization upon that of the previous one, so that the result of every search is a successive approximation of the global optimal

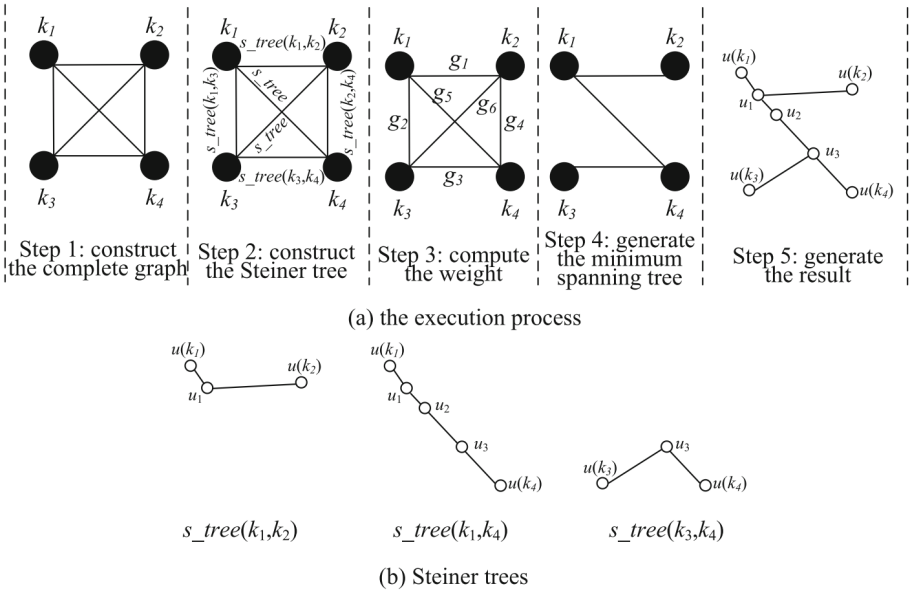

Figure 5.5. *An example of PACOKS execution for the query* $Q = \{k_1, k_2, K_3, k_4\}$ *(from [Lin et al., 2016])*

solution.

For each search, PACOKS build a complete graph G_K having the keywords $Q = \{k_1, ..., k_2\}$ as nodes. Then, for each edge $e(k_i, k_j) \in E(G_K)$, it get the minimum Steiner tree applying ACKOS with only the two keywords k_i and k_j. The weight of each Steiner tree is applied to the relative edge. The successive step is to compute the minimum spanning tree ST of G_K: each edge in ST is finally replaced with the relative Steiner tree producing the final solution. An example of the PACOKS steps is provided in Figure 5.5.

Conclusions

The answer building process is the core of the keyword search in databases. so that most of the effort spent in this field has been spent on this

component. Generally the solutions proposed follows the schema-based and graph-based approaches, and differentiate for the ranking functions used or the precautions introduced to allow a more efficient search. As a matter of fact, the time performance are the most critical issue that future works must address. The complexity of the problem prevents an efficient research. From this point of view, to precompute part of the workload may seem the only way to provide a keyword search system design to hit the market.

6 Evaluation

6.1 Evaluation in Information Retrieval

The 1958 witnessed the begin of the Cranfield Project [Cleverdon, 1997], lead by Cyril Cleverdon, namely the first scientific attempt to evaluate the different "retrieval systems". Even if the aim of this first *evaluation campaign* was to inspect manual library classification models, generally very distant from the current automated retrieval systems, the Cleverdon's work posed the basis for the information retrieval evaluation processes.

A systematic and comparable experimental evaluation of IR is a very demanding activity, both in terms of time and effort. For this reason, it is usually carried out in publicly open and large-scale evaluation campaigns at international level, which allow for sharing the effort, producing large experimental collections, and comparing state-of-the-art systems and algorithms [Bergamaschi et al., 2016]. In the years, several campaigns have been established, where the most important are the Text REtrieval Conference (TREC) in the United States (co-sponsored by the National

Institute of Standards and Technology - NIST), the Conference and Labs of the Evaluation Forum (CLEF) in Europe, and NII Testbeds and Community for Information access Research (NTCIR) in Japan and Asia.

The experimental method developed in the Cranfield tests has been highly influential, so that all these international evaluation activities rely on the *Cranfield methodology*: this evaluation procedure makes use of standard *test collections*, a triple composed by

- A **dataset**, a collection of documents where retrieve information
- A set of **topics** which simulates actual user information needs
- The **ground-truth**, a set of relevance judgments where for each topic the documents relevant for the topic are determined.

Evaluation campaigns promote the re-use of the experimental data and the acquired knowledge, giving the possibility to conduct studies to track the improvement on performances, to reproduce the obtained results, and to develop new solutions.

The effectiveness of evaluation campaigns is documented: TREC committee assert that the effectiveness of information retrieval systems doubled within six years since the beginning of the campaign[1]. Moreover, as reported by [Tassey et al., 2010], there is an economical benefit from these campaigns: for every $1 that NIST and its partners invested in TREC, at least $3.35 to $5.07 accrued to researchers and industry.

[1] `http://trec.nist.gov/overview.html`

6.2 Evaluation in Relational Keyword Search

The lack of a shared and complete evaluation methodology for relational keyword search systems is one of the main topics that [Bergamaschi et al., 2016] addresses. Without it, the development of new and efficient systems, designed to daily use, would not be possible in short times.

As for traditional IR systems, an empirical assessment of keyword-based retrieval systems is imperative [Webber, 2010]. Keyword search in databases share with traditional IR the task of providing results retrieved to fill the ambiguous information need expressed though the query, so that an empirical evaluation of the answers resulted by the systems is necessary to measure the effectiveness of the research from the user viewpoint. Moreover, due to the complexity of the task, there is also the necessity to systematically evaluate the keyword search systems from a time and memory consumption point of view.

At the current state, a systematic comparison of the current systems is hardly feasible due to the lack of an unified architectural approach that considers all of the issues of keyword search, from the interpretation of the user needs, to the computation, retrieval, ranking and presentation of the results. For example, it is hard to compare different systems that follow different approaches, such as schema- or graph-based, proposing different solutions for all the components of the pipeline treated in this thesis.

From this point of view, in [Bergamaschi et al., 2016] the authors address the need of a conceptual keyword search architecture pivoting around two different components:

- The *system-oriented* component, aiming to improve the performances and the *efficiency* of the search
- The *user-oriented*, aiming to improve the *effectiveness*, i.e. the quality of the search from a user perspective.

This two complementary aspects have been addressed individually in many evaluations, even though the user-oriented component has been less considered. In Table 6.1 we summarize the different datasets and query sets used in literature to evaluate and test the systems. These evaluation process and resources presented the following critical points:

- The evaluation and comparison of the systems are performed exploiting arbitrary datasets. It is trivial that different datasets produce different results, and that same datasets produce equivalent results. In keyword search literature, a plethora of different datasets has been used to evaluate the systems and, moreover, different evaluations use different subschemas of the original dataset. The practice of cropping the datasets reduces the dimensions and allows a faster execution, but at the same time alters the results, masking scalability issues and artificially bolstering the reported effectiveness of the system.
- The datasets dimensions and complexity are rarely provided, and whenever provided, represents small and simple databases.
- Systems are generally evaluated with efficiency benchmarks, lacking an effectiveness evaluation. Both should be considered, because a fast systems that result poor answers is useless.

- There is no uniformity when creating query: some authors build them picking random terms among the dataset, other researchers build queries that reflect their proposed ranking scheme. Moreover, self-authored queries have a strong potential for bias: it is too easy to formulate queries that are favorable to your own over other algorithms [Webber, 2010].

- The definition of relevance is often vague among the authors, and in the case of SPARK [Luo et al., 2008], which judges relevant a result containing all the query terms and have the smallest size of any result satisfying the first criterion, it is also in contrast to the definition of the IR community, i.e. a relevant results must address the underlying information need and not just contain all search terms. [Webber, 2010]. Furthermore, like self-authoring, self-assessment has the potential for biasing result.

- Finally, systems appear to perform abnormally well with regard to effectiveness metrics, with [Kacholia et al., 2005] authors claiming a near 100% of *recall* and EASE [Li et al., 2008b] authors a precision of 0.9.

Looking at the Table 6.1 we can see that the community converged to the use of a restricted number of recurring datasets. To understand the peculiarities of each one, we provide their characteristics, even thou we cannot give specifications like size and schema, due to the copious customizations applied by the researchers.

- TPC[2] is a non-profit corporation that provide transaction processing

[2] `http://www.tpc.org/`

System	Database			Queries			Performance		
	Name(s)	# Rel.	MB	Origins	# Queries	K. range	Mem.	Time	Eff.
DBXplorer	TPC-H, 3 Custom DB	19-84	$\geq$ 10	RDM	500/500/500/500	2-10	✓	✓	✓
DISCOVER	TPC-H (3 evaluations)	ND	$\geq$ 100	RDM	ND/100/200	2-5		✓	
DISCOVER II	DBLP	6	56	RDM	100/ND	2-5		✓	
PRECIS	IMDB	ND	ND	RES	140	ND		✓	✓
EFFECTIVE	Custom	5	ND	QL	50	6.7			✓
SPARK	DBLP, IMDb, MONDIAL	ND	ND	RES	18/22/35	2-4		✓	✓
LABRADOR	3 Custom DB	1/3/3	3/4/6	QL	33/15/34	1-4		✓	
MeanKS	TPC-E (only tested)	ND	ND	ND	ND	ND			
POWER	DBLP, IMDb	4/8	ND	RES	17/20	3-5		✓	
SQAK	Custom/TPC-H	6/8		RES	15/15	ND			✓
KEYMANTIC/KEYRY	Custom/IMDB	ND	ND	USR	29	ND		✓	✓
BANKS	DBLP, Custom	ND	ND	RES	7	1-2	✓	✓	
BANKS II	DBLP, IMDb	ND	ND	RDM	200	2-7		✓	
BLINKS	DBLP, IMDb	ND	ND	RES	60/40	2-8	✓	✓	
Golenberg et al.	MONDIAL	ND	ND	RDM	36	2-10		✓	
DPBF	DBLP, MoveLens	ND	ND	RDM	500/500/100	2-6		✓	
EASE	DbLife, DBLP, IMDb	ND	ND/400/5	RES	5/5	2-5		✓	✓
STAR	DBLP, IMDb, YAGO	ND	ND	RDM	180/180/120	3-7		✓	✓
PruneDP	DBLP, IMDb	ND	ND	RDM	50/50	ND	✓	✓	
PACOKS	DBLP	ND	800	RES	50	2-6		✓	
Dalvi	DBLP, IMDb	ND	99/94	RES	8/4	2-6		✓	
Ekso	TPC-H, Custom	ND	500/929	RDM	50/50	2-4	✓	✓	✓
SAINT	DBLP, IMDb	4/4	470/ND	QL	100/100	ND	✓	✓	

Table 6.1. *List of the the systems presented in this thesis with the datasets and query sets used do evaluate the performance or the effectiveness of the search. Legend - ND: Not Defined; RDM: query random taken from the dataset; RES: query posed by researchers; USR query posed by users; QL: query built from DBMS query logs; Mem: Memory; Eff: Efficiency.*

and database benchmarks. The TPC-E dataset models the activity of brokerage firm that must manage customer accounts, execute customer trade orders, and be responsible for the interactions of customers with financial markets, while TPC-H models the content of an industry which must manage sell, or distribute products worldwide.

- The DBLP[3] computer science bibliography is the on-line reference for bibliographic information on major computer science publications, containing more than 3.3 million publications and more than 1.7 million authors.

- IMDB[4] is the world's most popular and authoritative source for movie, TV and celebrity contents. Its database contains more than 185 million data items including more than 3.5 million movies, TV and entertainment programs and 7 million cast and crew members.

- MONDIAL[5] dataset comprises geographical and demographic information from the CIA World Factbook, the International Atlas, the TERRA database, and other web sources.

Two of the datasets reported above have been used in [Coffman and Weaver, 2010] to define an unified evaluation framework, designed to be a first pass to overcome the lack of a systematic evaluation process in the keyword search field. This framework is based on a Cranfield methodology triple composed by:

[3] http://dblp.uni-trier.de/

[4] http://www.imdb.com/

[5] https://www.dbis.informatik.uni-goettingen.de/Mondial/

- Datasets: they proposed two datasets built from subsets of the IMDb and Wikipedia databases (the latter is a selection of articles crawled from the website[6]) and the entire MONDIAL. Albeit MONDIAL is smaller in size than the others, its schema is much more complex. The characteristics of the datasets are represented in Table 6.2.

- Topics: the authors materialize topics as keyword queries. They produce fifty queries (the traditional minimum for evaluation purpose), paying attention to: (a) not produce redundant information need, (b) not produce ambiguous queries, (c) produce queries for the domain-specific datasets.

  All the queries have been produced by the authors, differently form what happens in evaluation campaigns, where a number of individuals create candidate information needs from which only a small subset is actually chosen. This procedure aims to avoid biased sets of queries, but is impractical for keyword search in databases due to the lack of incentive for other research to participate in the campaign. Queries statistics are summarized in Table 6.3.

- Relevance assessment: for all the information needs, the authors identify relevant results by constructing the information needs around a template of database relations. Then, a number of SQL queries are posed to the DBMSs to identify all possible results satisfying the information need and judge each of these results for relevance. They use binary relevance assessments when judging results.

[6] https://www.wikipedia.org/

Dataset	Size (MB)	Relations	Tuples
MONDIAL	9	28	17,115
IMDb	516	6	1,673,074
Movie (**id**, title, year)			181,706
Person (**id**, name)			273,034
Character (**id**, name)			206,951
Role (**id**, type)			11
Cast (***movieId***, ***personId***, ***characterId***, ***roleId***)			812,694
MovieInfo (**id**, *movieId*, info)			198,678
Wikipedia	550	6	206,318
Page (**id**, title)			5,540
Revision (**id**, *pageId*, *textId*, *userId*)			5,540
Text (**id**, text)			5,540
User (**id**, name)			1,745
PageLinks (**id**, *from*, *to*)			187,951
UserGroups (***userId***, **group**)			2

Legend **primary key**, *foreign key*, full text index

Table 6.2. *Characteristics and simplified schema of the datasets used in Coffman and Weaver's framework. The reported size includes database indices. (Taken form [Coffman and Weaver, 2010])*

	Search log [26]	Synthesized			Results	
Dataset	$\overline{[\![q]\!]}$	$\lvert Q\rvert$	$[\![q]\!]$	$\overline{[\![q]\!]}$	$[\![R]\!]$	$\overline{[\![R]\!]}$
MONDIAL		50	1–5	2.04	1–35	5.90
IMDb	2.71	50	1–26	3.88	1–35	4.32
Wikipedia	2.87	50	1–6	2.66	1–13	3.26
Overall	2.37	150	1–26	2.86	1–35	4.49

Legend
$\lvert Q\rvert$ total number of queries
$[\![q]\!]$ range in number of query terms
$\overline{[\![q]\!]}$ average number of terms per query
$[\![R]\!]$ range in number of relevant results per query
$\overline{[\![R]\!]}$ average number of relevant results per query

Table 6.3. *Query and result statistics of Coffman and Weaver framework. (Taken form [Coffman and Weaver, 2010])*

6.3 Effectiveness Evaluation

In [Coffman and Weaver, 2010] is presented the first systematic comparison between keyword search systems to explicitly use the above mentioned framework to evaluate them. They compare the 8 state-of-the-art systems indicated in Table 6.4 with the $*$ sign.

Before this work, the singular systems have been compared to a limited set of other systems on an arbitrary subset of datasets. The non standard datasets, along with the lack of cross evaluation, makes difficult to compare the trade-offs between approaches that vary widely in both query processing and ranking results. In Table 6.4 we summarize the the comparisons realized in literature. As expected, systems with similar approaches are more easily to compare, e.g. the BANK I-II systems have been widely used as a benchmark for graph-based systems. It worth noticing that several comparison between systems with different approach have been done. Generally, these comparisons do not take in considerations all the systems aspects, and are performed thought expedients used to adapt the systems to the situations.

To measure the effectiveness of search systems, the authors use four metrics derived from the information retrieval field.

- **Number of top-1 relevant results** is the number of queries for which the first result is relevant
- **Reciprocal rank** is the reciprocal of the highest ranked relevant result for a given query
- **Average Precision (AP)** is the average of the precision values

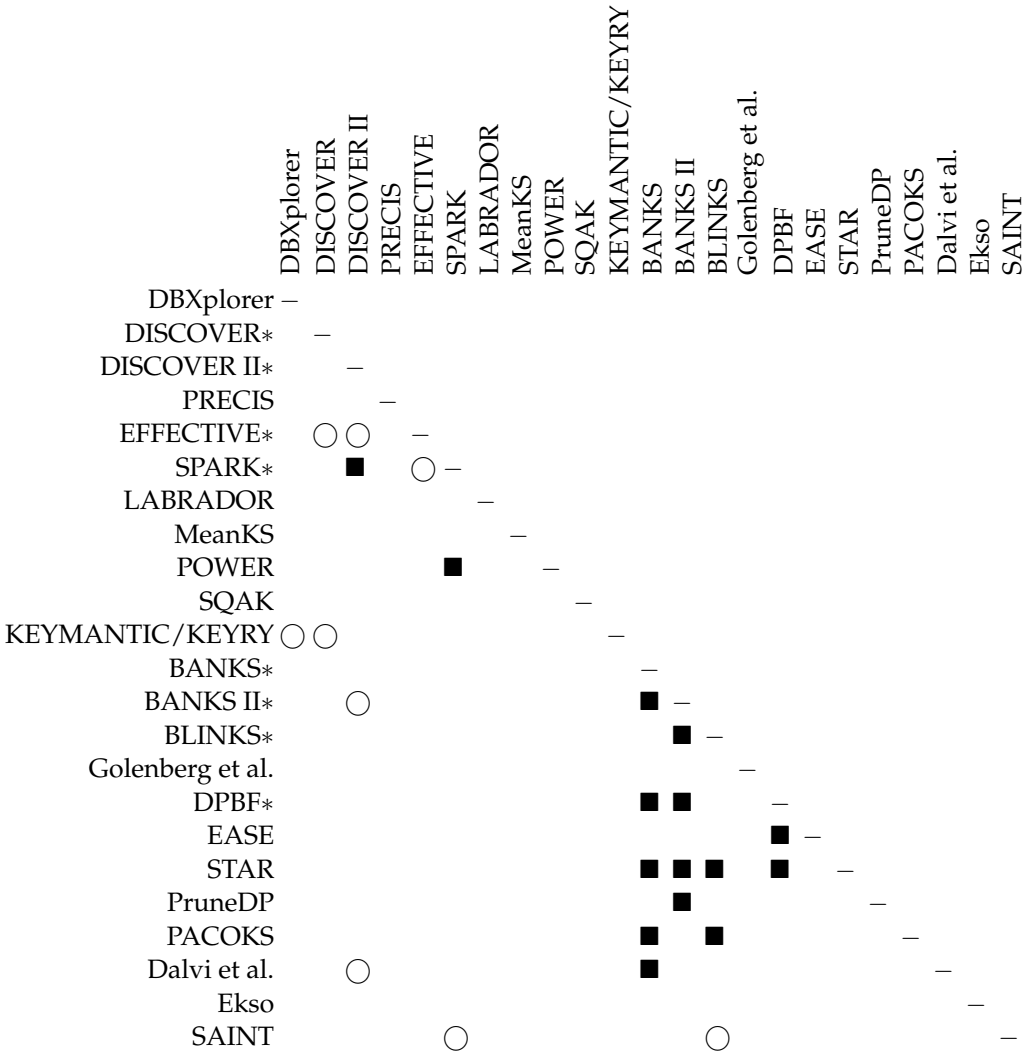

Table 6.4. *System evaluation comparison matrix. On the left are listed the systems that have been compared with the systems in the header.* ■ *means that the evaluation is naturally feasible, while* ◯ *represents an approximate comparison, i.e. between systems of different approaches. Systems marked with* ∗ *have been compared using the C&W framework (Based and expanded from [Coffman and Weaver, 2010])*

calculated after each relevant result is retrieved (and assigning a precision of 0.0 to any relevant results not retrieved)

- **11-point interpolated average precision** is used to summarize the entire precision-recall curve

To calculate each metric, they retrieve the top 1000 results for each system.

The measurements of effectiveness realized by the authors are considerably lower than those reported in the previous evaluations. The trend toward reporting above-average effectiveness scores have been confirmed in [Webber, 2010], and possibly caused by the biased queries posed. Moreover, the scores of retrieval systems evaluated are much higher with respect to the systems evaluated at TREC and INEX.

Relative to the systems evaluated, most of the systems perform comparably on each dataset, while graph based are generally more effective than schema-based approaches. Furthermore, the results empirically proved that there is no need to prefer a system that exploits complex IR-style scoring functions and complicated processing algorithms over a simple structure ranking function,

The authors conclude noticing that PageRank-like concept plays an important factor when ranking results: systems pivoting around node weights as BANKS [Bhalotia et al., 2002] and Bidirectional [Kacholia et al., 2005] perform better than systems as DPBF [Ding et al., 2007] focused on minimizing the weight of the result tree.

6.4 Efficiency Evaluation

In continuity with the work discussed above, Coffman and Weaver conducted a performance analysis on seven systems (DISCOVER I-II, BANKS I-II, DPBF, STAR, SPARK), and provided their conclusion in [Coffman and Weaver, 2014].

The analysis were conducted exploiting the evaluation framework introduced above, and took in consideration the execution time and the memory consumption of the algorithms.

Execution Time

Typically the systems proposed do not address a *complete* search on the database, i.e. they do not aim to retrieve all the relevant results, but result only the top-k relevant results. The variables that affect the run-time performances are different: the retrieval depth, the number of search terms, the frequency of search terms in the database, the numbers of database tuples and the complexity of the schema. Aside from this, the authors found out that the performance of the systems evaluated were disappointing, particularly with regard to the number of queries completed successfully (they impose a maximum execution time of 1 hour for each search technique): existing search techniques provide reasonable performance only on the smallest dataset (MONDIAL). Performance degrades significantly when we consider a dataset with hundreds of thousands of tuples (Wikipedia) and becomes unacceptable for a data set with a million tuples (IMDb).

Memory Consumption

The schema-based systems consume very little memory, most of which is used for the database schema. In contrast, the graph-based approaches require considerably more memory to store their data graph. Even if never documented before, the graph-based approaches hardly contains the space used to store the initial data graph: significant is the case of BLINKS [He et al., 2007]: its bi-level index could occupy more than 160GB of memory to represents the IMDb dataset.

Due to excessive memory consumption (the total amount of memory was 5GB), several queries cannot be complete and lead to memory fault, unveiling the unreliable behavior of this systems.

6.5 Toward a Reference Evaluation Framework

Even if the Coffman and Weber framework represents an important step towards a fair evaluation of keyword search approaches, the authors of [Bergamaschi et al., 2016] pointed out four main limitations of this work:

- The adopted metrics to evaluate effectiveness and efficiency are biased in most of the schema-based systems, because a certain amount of time is required for the execution of the SQL queries by the DBMS underlying the application, that is independent form the retrieval algorithm, and depends from the underlying DBMS.

- The effectiveness measurements of schema-based systems is altered because of the intrinsic nature of the approach: it provides SQL queries as a primary result, so that all tuples resulting from the same

SQL query have intrinsically the same score, and that the same result can be obtained by different queries.

- Most of the queries in the dataset are composed of only one element, so that the evaluation do not test the algorithm at all
- The benchmark does not discuss what is a *correct* result in terms of granularity.

To address these issues, the authors proposed some guidelines to build an evaluation framework based on Cranfield paradigm tailored for keyword search over structured data.

- The dataset must be representative of the domain of interest both in terms of data and size, so that they have to be decided on the basis of the search task the system has to address. Furthermore, the dataset must also have a complex structure made of interconnected tables, to properly test the retrieval algorithm.
- The topics must simulate actual user information needs and could be prepared from real system logs, gathered by means of task-based analysis, or through a deep interaction with the involved stakeholders. As a consequence, the evaluation is conducted starting from the information need and not from ready-to-use query, as in numerous previous evaluations. Furthermore, it is necessary to define information needs that can be translated into queries composed of more than one keyword, for the reasons explained above.
- The ground-truth is essential to evaluate the effectiveness of a systems, and it is important that the relevance judgments are as most unbiased

as possible. For this reason the possible results to a query have to be judged by a pool of domain users that decide if a result is relevant for a given information need. To avoid the issue of schema-based systems that provide sets of equally rank tuples, the SQL queries could be be a good candidate for the evaluation for these systems. Nevertheless, to consider SQL queries as a result of a search system makes graph-based approaches not comparable with schema-based approaches.

Conclusions

It is necessary for the development of this research area that the community recognize the need of an unified and shared framework to evaluate the keyword search systems in a systematic. It has been demonstrated by the information retrieval evaluation campaigns that the creation of a tool for the systematical comparison and evaluation of the different approaches is the fist step in the right direction for a more objective and efficient improvement of the performances, both in terms of time and effectiveness. Without a comprehensive benchmark, it is impossible to identify the valuable system components and approach, leading to scientific research based on speculations.

7 Conclusions

In the last years, a lot of effort has been put on designing different systems implementing natural language search on relational databases. The community proposed integrated solutions built to efficiently accomplish the task, but they lacked a complete and general view on the issue, so that many critical aspects do not received the necessary attention.

With our work we propose a general pipeline for keyword search systems and we generalize the approaches proposed in literature, disclosing to the reader the state-of-the-art of the field. This pipeline takes in considerations all the aspects of a keyword search systems: we delineate four different elements, i.e. the data processing, the query processing, the matching step and the answers building. Each component is individually presented in the chapters of this thesis, because each component is equally important from a system viewpoint. These components are deeply integrated together, so that the peculiarities of each component depend from the specific system and from the others components.

The content of this thesis can be used as a first step toward the

development of new solutions, because although a lot of research has been done to present more and more efficient and effective solutions, there is still a great deal of work ahead to overcome the proof-of-concept state of these works and to present a real commercial system designed for a daily use. The most of the research in this field aimed to enhance the efficiency of the keyword search systems, overlooking the components that do not directly affect this aspect. Differently, we analyzed each component details, providing a close examination of the systems. In Table 7.1 we synthetically present the characteristics of the main systems studied and analyzed in this thesis. We take into account each component, showing the approaches followed by the authors. The contents of this thesis, resumed in the table, are exposed in the following paragraphs.

The *data processing* component aims to manage and organize the data contained in the database to allow the successive search, materializing indexes and other auxiliary structures. As we can see from Table 7.1, most of the systems implement the schema- or graph-based approaches (SCH, GRA in the table, respectively) to represent the database content. The schema graph G_S materialize the database as a network of relations, while the data graph G_D as a network of tuples. G_S is than light and flexible compared to G_D: from this viewpoint, the latter systems suffer from critical scalability issues that could undermine the feasibility of this kind of approach due to the potential large dimensions of the data graph. The Multi-Granular-Graph (M-GRA) approach of Dalvi et al. [Dalvi et al., 2008] aims to resolve these memory consumption issues materializing in RAM only a small part of the entire data-graph, while the rest could be

System	Data		Query		Answers	
	Repres.	Index/Matching	Semantics	Q. processing	Structure	Ranking
DBXplorer	SCH	Symbol Table	$\wedge$	-	SQL queries	Number of joins
DISCOVER	SCH	Master Index	$\wedge$	full-text DBMS	SQL queries	Number of joins
DISCOVER II	SCH	Master Index	$\wedge$, $\vee$	full-text DBMS	SQL queries	IR at attribute level
PRECIS	SCH	Master Index	$\wedge$, $\vee$, $\neg$	full-text DBMS	DB Subset	G_S Edge Weights
EFFECTIVE	SCH	Master Index	$\wedge$, $\vee$	full-text DBMS	SQL queries	IR at attribute level
SPARK	SCH	Master Index	$\wedge$, $\vee$, $\neg$, WC	full-text DBMS	SQL queries	IR at attribute level
LABRADOR	SCH	Master Index	$\wedge$	full-text DBMS	SQL queries	Similarity BN
MeanKS	SCH	Master Index	$\wedge$	full-text DBMS	SQL queries	G_S Edge Weights
POWER	SCH	Master Index	$\wedge$	full-text DBMS	Tuple structures	ND
SQAK	SCH	Inverted Index	$\wedge$	–	SQL queries	G_S Node/Edge Weights
KEYMANTIC/ KEYRY	SCH	–	$\wedge$	–	SQL queries	ND
Golenberg et al.	GRA	ND	$\wedge$, $\vee$	–	Tuples trees	Proximity/Node Prestige
BANKS	GRA	Double index	$\wedge$	–	Tuples trees	Proximity/Node Prestige
BANKS II	GRA	Double index	$\wedge$	–	Tuples trees	Proximity/Node Prestige
BLINKS	GRA	Bi-level index	$\wedge$	–	Tuples trees	Proximity/Node Prestige
DPBF	GRA	Not specified	$\wedge$	–	Tuples trees	Proximity/Node Prestige
EASE	GRA	Extended index	$\wedge$	–	r-radius graphs	IR / Proximity
STAR	GRA	Double index	$\wedge$	–	Tuples trees	Proximity/Node Prestige
PruneDP	GRA	Double index	$\wedge$	–	Tuples trees	G_D Weights
PACOKS	GRA	Double index	$\wedge$	–	Tuple subgraphs	Proximity/Node Prestige
Dalvi	M-GRA	M-GRA	$\wedge$	–	Tuples trees	Proximity/Node Prestige
EKSO	VD	Inverted Index	$\wedge$	full-text DBMS	Virtual Documents	full-text DBMS
SAINT	VD	SKSA/KPSA	$\wedge$	ND	Virtual Documents	IR / proximity

Table 7.1. *A overview of most of the systems surveyed in this work with their characteristics. Legend: SCH: Schema-based; GRA: Graph-based; M-GRA: Multi-Granular-Graph; VD: Virtual Documents; –: not implemented; ND: not defined; $\wedge$: AND; $\vee$: OR; $\neg$: NOT; WC: Wildcards.*

disk-resident. Finally, a totally different approach is represented by the virtual document (VD) approach, that materializes plain documents built offline from the information contained in the database instance, in order to efficiently retrieve this documents with traditional IR techniques.

The *matching process* generally uses precomputed indexes to efficiently match the query keywords with the database elements. Researchers adopt structures and techniques derived from the IR inverted index and indexing process to do the task. In the literature, the indexing details (as for the query processing details) are generally omitted either because indexing process is entrusted to the DBMS or because effectiveness issues are not the main focus. In particular, the schema-based approaches generally use Master Indexes assembled from the inverted indexes built on each relation attribute using the DBMS indexing capabilities. Differently, in DBXplorer [Agrawal et al., 2002] the authors designed their own indexing system: the Symbol Table can handle different location granularities, so that it can index the rows, the columns or the cells of each table; SQAK [Mesquita et al., 2007] exploits Apache Lucene for indexing the content of each relation attribute, while Keymantic and Keyry [Bergamaschi et al., 2011a,b] do not rely on any index to do the matching between the keywords and the database elements.

The most of the graph-based systems implement a double index, where the first one maps each database term with the RowIDs of the tuples that contain the term, while the second one maps RowIDs with G_D nodes. Indexes are used in BLINKS [He et al., 2007] both to allow the matching step and to accelerate the answers building process: it materialize a bi-level graph storing the minimum distances between the nodes on the graph. In

EASE [Li et al., 2008b], the authors implement an extended index that maps the heterogeneous data source handled by the system. [Golenberg et al., 2008] and DPBF [Ding et al., 2007] do not provide any information on the indexing process adopted.

The virtual document approach exploits inverted index derived from the IR field. In particular, SAINT [Jianhua Feng et al., 2011] proposed two solutions: the single-keyword-based structure-aware index maps each term with the related virtual documents, while the keyword-pair based structure-aware index stores, for each couple of keywords, a pre-computed mutual score that would eventually been computed using the SKSA index.

The *Query processing* component aims to enhance the effectiveness of the search applying technique designed to sanitize the query and provide a more effective matching. Even though most of the components of the information retrieval query processing could be easily adopted in this field, e.g. stopword removing and stemming, little attention has been payed on it. In Table 7.1 we can see that schema-based systems generally entrust the DBMS full-text features to process the query (many of them did not specify the implementation details), whereas graph-based solutions do not even address the problem. As a matter of fact, the aim of the current generation of systems focuses on performance and efficiency, while lacking any detail on marginal (from this point of view) topics. This assumption implies that a commercial implementation of a keyword search system is still far from to be released. In Table 7.1 we also specify the logical semantics applied to the queries. Most of the systems apply the AND semantics in order to output only the answers containing all the keywords of the query.

Generally, applying the OR semantics allows to obtain more relevant results, but increase the computational complexity because the algorithms must take in account all the possible keyword configurations. Rarely the systems use a more rich semantic.

The major contributions in the keyword search field have been produced to design and lighten the *answers building process*. The structures built from the database representation graph, either from G_S or G_D, are expensive to be computed and their nature depends on both the graph structure and algorithm used. The approach generally followed is to calculate the top-k ranked results, avoiding to waste time to calculate results that the user would not consult. This results are produced on-the-fly inspecting the graphs, and are ranked according to functions that take in considerations both the structure topology and the IR features.

The schema-based systems generally produce Minimal Total Join Networks of Tuples (MTJNTs) through the analysis of the relations content. These networks are applied to patterns to produce structured SQL queries. Differently, In [Zeng et al., 2016]the authors designed a system resulting tuple structures built exploiting only the DBMS and without an on-the-top system. The task of the system proposed in [Simitsis et al., 2007] differs from the other approaches, because it is designed to result database subgraphs containing all the information related to the content of the query. The first naive ranking functions of DBXplorer [Agrawal et al., 2002] or Discover [Hristidis and Papakonstantinou, 2002] only takes in account the dimensions of the MTJNTs, assuming that more compact networks are more significant than large ones. DISCOVER II [Hristidis et al., 2003] added an

IR-fashion ranking function, based on tf-idf measures. However, it has been empirically demonstrated in [Coffman and Weaver, 2010] that this kind of ranking functions does not provide any significant effectiveness enhancement with respect to ranking functions that only take in account structural factors. As a matter of fact, most of the systems rank the answers with a score proportional to the sum of weight of nodes and edges on the graph, computed in heterogeneous ways among the different cases. In LABRADOR [Mesquita et al., 2007] the authors trained a Bayesian Network to compute a measure of similarity between the SQL queries resulted and the original query, in order to provide the ranked list.

Generally, the graph-based systems rely on building connected trees from the graph nodes, applying algorithms to solve or approximate the Steiner Tree Problem. These solutions generally rank their results using proximity-based score emphasizing compact results, designed to consider the PageRank-style prestige of the nodes. Differently, EASE materialize his answers as r-radius graphs, a solution suited to represent the heterogeneous data that the system handles. The score of this answers consider both IR measures and structural factors.

The complexity of the problem prevents the algorithm to produce results in acceptable times, even with small databases. In our opinion, this is the most relevant problem of the current generation of keyword search systems. The performance of graph- and schema-based solutions encouraged researchers to design the virtual document-based systems, that result ranked lists of virtual document, which scores are computed using traditional IR techniques. In the case of SAINT [Jianhua Feng et al., 2011], the virtual documents are connected in a graph through links built

on shared tuples, so that the system could retrieve aggregates of virtual documents, which scores are computed using proximity measures. In our opinion, designing systems that lighted the load of the algorithm pre-computing partial results is necessary for the systems of the future.

Finally, we could not propose a thesis written to be a first step toward the development of new keyword search over databases solutions without analyzing the situation over the scientific evaluation of these systems. After years of development in this field, it is now time for the community to spend effort on building a complete and systematical evaluation framework, following the examples of the information retrieval evaluation campaigns. This step cannot be further postponed, because new proposed solutions must be evaluated objectively, avoiding the biased results that the researchers got from the comparisons done until now. New standard collections, topics and judgments must be proposed by large dedicated pools of researchers: the future of the keyword search field depends on valuable results, and these results can be achieved only through a shared, standard and systematical evaluation process.

Bibliography

Abiteboul, S., Hull, R., and Vianu, V. (1995). *Foundations of databases*. Addison-Wesley, Reading, Mass.

Agrawal, S., Chaudhuri, S., and Das, G. (2002). DBXplorer: a system for keyword-based search over relational databases. In *Proceedings of the 18th International Conference on Data Engineering*, pages 5–16.

Bergamaschi, S., Domnori, E., Guerra, F., Orsini, M., Lado, R. T., and Velegrakis, Y. (2010). Keymantic: semantic keyword-based searching in data integration systems. In *Proceedings of the VLDB Endowment*, volume 3, pages 1637–1640.

Bergamaschi, S., Domnori, E., Guerra, F., Trillo Lado, R., and Velegrakis, Y. (2011a). Keyword search over relational databases: a metadata approach. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 565–576.

Bergamaschi, S., Ferro, N., Guerra, F., and Silvello, G. (2013a). Keyword search and evaluation over relational databases: an outlook to the future.

In *Proceedings of the 7th International Workshop on Ranking in Databases*, page 8.

Bergamaschi, S., Ferro, N., Guerra, F., and Silvello, G. (2016). Keyword-Based Search Over Databases: A Roadmap for a Reference Architecture Paired with an Evaluation Framework. In *Transactions on Computational Collective Intelligence XXI*, pages 1–20. Springer.

Bergamaschi, S., Guerra, F., Interlandi, M., Trillo-Lado, R., and Velegrakis, Y. (2013b). QUEST: a keyword search system for relational data based on semantic and machine learning techniques. *Proceedings of the VLDB Endowment*, 6(12):1222–1225.

Bergamaschi, S., Guerra, F., Rota, S., and Velegrakis, Y. (2011b). A hidden markov model approach to keyword-based search over relational databases. In *International Conference on Conceptual Modeling*, pages 411–420. Springer.

Bhalotia, G., Hulgeri, A., Nakhe, C., Chakrabarti, S., and Sudarshan, S. (2002). Keyword searching and browsing in databases using BANKS. In *Proceeding of the 18th International Conference on Data Engineering*, pages 431–440.

Bourgeois, F. and Lassalle, J.-C. (1971). An Extension of the Munkres Algorithm for the Assignment Problem to Rectangular Matrices. *Commununications of the ACM*, 14(12):802–804.

Brin, S. and Page, L. (1998). The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30.

Carpineto, C. and Romano, G. (2012). A Survey of Automatic Query Expansion in Information Retrieval. *ACM Computing Surveys*, 44(1):1–50.

Cleverdon, C. (1997). The Cranfield tests on index language devices. In Sparck Jones, K. and Willett, P., editors, *Readings in Information Retrieval*, pages 47–59. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387.

Coffman, J. and Weaver, A. C. (2010). A framework for evaluating database keyword search strategies. In *Proceedings of the 19th ACM international conference on Information and knowledge management*, pages 729–738.

Coffman, J. and Weaver, A. C. (2014). An Empirical Performance Evaluation of Relational Keyword Search Techniques. *IEEE Transactions on Knowledge and Data Engineering*, 26(1):30–42.

Croft, B., Metzler, D., and Strohman, T. (2009). *Search Engines: Information Retrieval in Practice*. Addison-Wesley Publishing Company, USA, 1st edition.

Dalvi, B. B., Kshirsagar, M., and Sudarshan, S. (2008). Keyword search on external memory data graphs. *Proceedings of the VLDB Endowment*, 1(1):1189–1204.

Demidova, E., Fankhauser, P., Zhou, X., and Nejdl, W. (2010). DivQ: diversification for keyword search over structured databases. In *Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval*, pages 331–338.

Ding, B., Yu, J. X., Wang, S., Qin, L., Zhang, X., and Lin, X. (2007). Finding top-k min-cost connected trees in databases. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 836–845.

Furnas, G. W., Landauer, T. K., Gomez, L. M., and Dumais, S. T. (1987). The vocabulary problem in human-system communication. *Communications of the ACM*, 30(11):964–971.

Golenberg, K., Kimelfeld, B., and Sagiv, Y. (2008). Keyword proximity search in complex data graphs. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 927–940.

Guha, R., McCool, R., and Miller, E. (2003). Semantic search. In *Proceedings of the 12th international conference on World Wide Web*, pages 700–709.

He, H., Wang, H., Yang, J., and Yu, P. S. (2007). BLINKS: ranked keyword searches on graphs. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, pages 305–316.

Hristidis, V., Gravano, L., and Papakonstantinou, Y. (2003). Efficient IR-style keyword search over relational databases. In *Proceedings of the 29th International Conference on Very Large Data Bases*, volume 29, pages 850–861.

Hristidis, V. and Papakonstantinou, Y. (2002). Discover: Keyword search in relational databases. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 670–681. Proceeding of the 2002 VLDB Endowment.

Hwang, F. and Richards, D. S. (1992). Steiner tree problems. *Networks*, 22(1):55–89.

Jianhua Feng, Guoliang Li, and Jianyong Wang (2011). Finding Top-k Answers in Keyword Search over Relational Databases Using Tuple Units. *IEEE Transactions on Knowledge and Data Engineering*, 23(12):1781–1794.

Kacholia, V., Pandit, S., Chakrabarti, S., Sudarshan, S., Desai, R., and Karambelkar, H. (2005). Bidirectional expansion for keyword search on graph databases. In *Proceedings of the 31st International Conference on Very Large Data Bases*, pages 505–516.

Kargar, M., An, A., Cercone, N., Godfrey, P., Szlichta, J., and Yu, X. (2014). MeanKS: meaningful keyword search in relational databases with complex schema. pages 905–908.

Kargar, M., An, A., Cercone, N., Godfrey, P., Szlichta, J., and Yu, X. (2015). Meaningful keyword search in relational databases with large and complex schema. In *Proceedings of the 31st International Conference on Data Engineering*, pages 411–422.

Kasneci, G., Ramanath, M., Sozio, M., Suchanek, F. M., and Weikum, G. (2009). STAR: Steiner-Tree Approximation in Relationship Graphs. In *Proceeding of the 2009 International Conference on Data Engineering*, pages 868–879.

Kumar, R. and Tomkins, A. (2009). A Characterization of Online Search Behavior. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*.

Li, G., Feng, J., and Zhou, L. (2008a). Retune: Retrieving and materializing tuple units for effective keyword search over relational databases. In *Proceedings of the 27th International Conference on Conceptual Modeling*, pages 469–483. Springer.

Li, G., Ooi, B. C., Feng, J., Wang, J., and Zhou, L. (2008b). EASE: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 903–914.

Li, L., Shang, Y., Shi, H., and Zhang, W. (2002). Performance evaluation of hits-based algorithms. In *Communications, internet, and information technology*, pages 171–176.

Li, R.-H., Qin, L., Yu, J. X., and Mao, R. (2016). Efficient and Progressive Group Steiner Tree Search. pages 91–106.

Lin, Z., Xue, Q., and Lai, Y. (2016). Pacoks: Progressive ant-colony-optimization-based keyword search over relational databases. In *International Conference on Web-Age Information Management*, pages 107–119. Springer.

Liu, F., Yu, C., Meng, W., and Chowdhury, A. (2006). Effective keyword search in relational databases. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 563–574.

Liu, S., Liu, F., Yu, C., and Meng, W. (2004). An effective approach to document retrieval via utilizing WordNet and recognizing phrases. In *Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 266–272.

Luo, Y., Lin, X., Wang, W., and Zhou, X. (2007). Spark: top-k keyword query in relational databases. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 115–126.

Luo, Y., Wang, W., and Lin, X. (2008). SPARK: A Keyword Search Engine on Relational Databases. In *Proceedings of the 24th International Conference on Data Engineering*, pages 1552–1555.

Markowetz, A., Yang, Y., and Papadias, D. (2007). Keyword search on relational data streams. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, pages 605–616.

Mesquita, F., da Silva, A. S., de Moura, E. S., Calado, P., and Laender, A. H. (2007). LABRADOR: Efficiently publishing relational databases on the web by using keyword-based query interfaces. *Information Processing & Management*, 43(4):983–1004.

Nandi, A. and Jagadish, H. (2009). Qunits: queried units in database search. *arXiv preprint arXiv:0909.1765*.

Park, J. and Lee, S.-g. (2011). Keyword search in relational databases. *Knowledge and Information Systems*, 26(2):175–193.

Qin, L., Yu, J. X., and Chang, L. (2009). Keyword search in databases: the power of RDBMS. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 681–694.

Simitsis, A., Koutrika, G., and Ioannidis, Y. (2007). Précis: from unstructured keywords as queries to structured databases as answers. *The VLDB Journal*, 17(1):117–149.

Singh, J. and Gupta, V. (2016). Text Stemming: Approaches, Applications, and Challenges. *ACM Computing Surveys*, 49(3):1–46.

Su, Q. and Widom, J. (2005). Indexing relational database content offline for efficient keyword-based search. In *Proceedings of the 9th International Database Engineering & Application Symposium (IDEAS'05)*, pages 297–306.

Tassey, G., Rowe, B. R., Wood, D. W., Link, A. N., and Simoni, D. A. (2010). Economic impact assessment of NIST's text REtrieval conference (TREC) program. *National Institute of Standards and Technology, Gaithersburg, Maryland*.

Tata, S. and Lohman, G. M. (2008). SQAK: doing more with keywords. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 889–902.

van Rijsberben, C. J. (1979). *Information Retrieval*. Butterworth-Heinemann Newton, MA, USA.

Webber, W. (2010). Evaluating the effectiveness of keyword search. *IEEE Data Engineering Bulletin*, 33(1):54–59.

Yang, X., Procopiuc, C. M., and Srivastava, D. (2011). Summary graphs for relational database schemas.

Yi Luo, Wei Wang, Xuemin Lin, Xiaofang Zhou, Jianmin Wang, and Kequi Li (2011). SPARK2: Top-k Keyword Query in Relational Databases. *IEEE Transactions on Knowledge and Data Engineering*, 23(12):1763–1780.

Yu, J. X., Qin, L., and Chang, L. (2010). Keyword Search in Relational Databases: A Survey. *IEEE Data Eng. Bull.*, 33(1):67–78.

Zeng, Z., Lee, M. L., and Ling, T. W. (2016). PowerQ: An Interactive Keyword Search Engine for Aggregate Queries on Relational Databases. In *Proceedings of the 19th International Conference on Extending Database Technology*. OpenProceedings.

Zobel, J. and Moffat, A. (2006). Inverted files for text search engines. *ACM computing surveys*, 38(2):6.